Università degli Studi di Pisa
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Analisi delle performance di Spark tramite tecniche di Data Warehouse

Relatore: Prof. Antonio Cisternino

Presentata da:
Federico Conoci

Sessione Dicembre
Anno Accademico 2016/17

UNIVERSITÀ DI PISA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Analisi delle performance di Spark tramite tecniche di Data Warehouse

Tesi di

Federico Conoci

Relatore: Prof. Antonio Cisternino .....

Presentata da:

Federico Conoci .....

Sessione Laurea Dicembre 2016

Anno Accademico 2016/2017

Indice

Introduzione

Si stima che nel 1992 sono stati generati cento GB di dati al giorno, dieci anni dopo la stessa quantità viene generata ogni secondo e attualmente le valutazioni indicano che vengono generati due quintilioni e mezzo ($2^{30} \times 5^{30}$) di bytes all'anno[1]. Si pensi che il 90% dei dati nel mondo sono stati creati negli ultimi due anni. Questa mole impressionante di dati è destinata a crescere, infatti, si prevede che nel 2018 verranno generati 50.000 GB ogni secondo. Queste informazioni provengono da tutto il mondo e sono ottenute dalle fonti più diverse: il New York Stock Exchange genera tra i 4 e i 5 terabytes quotidianamente, Facebook ospita più di 240 miliardi di foto, il Large Hadron Collider in svizzera produce circa 30 petabytes di dati all'anno.

L'incremento della produzione dei dati, sia in termini di volume che velocità, ha portato alla necessità di creare nuove tecnologie che fossero in grado di estrarre e analizzare quelli che oggi vengono chiamati *Big Data*. I *Big Data* hanno avuto un forte impatto non solo nel mondo informatico ma anche nella società. Infatti, la possibilità di raccogliere una immensa quantità e varietà di informazioni ha assunto sempre più valore. Questo perchè l'importanza delle informazioni ricavabili dai Big Data sta nel fatto che posssono essere utilizzate nei settori più vari con un grande riscontro oggettivo.

Negli ultimi vent'anni la maggior parte delle applicazioni analitiche sono state create utilizzando dati strutturati estratti da sistemi che venivano poi consolidati su *Data Warehouse*. La particolarità dei *Big Data* è invece contraddistinta dalle quattro "V": volume dei dati, velocità di analisi, varietà delle sorgenti e veridicità. I sistemi tradizionali non sono quindi in grado di gestire in modo efficiente questi aspetti ed è per questo che sono state sviluppate nuove tecnologie. Le soluzioni che si sono affermate sono Apache

Hadoop e Spark. Hadoop dell'Apache Software Foundation è un *framework* per applicazioni che girano su cluster *commodity*. Questo *framework* include un *file system* distribuito in grado di gestire e distribuire elevati volumi di dati attraverso i nodi del cluster; utilizzando il modello di programmazione MapReduce è in grado di frammentare le elaborazioni su più nodi e fare quindi leva sull'elaborazione parallela. Spark è una piattaforma per il calcolo parallelo e distribuito che assolve molte delle funzioni di Hadoop integrandosi con quest'ultimo e con il suo ecosistema.

Da un punto di vista delle performance Spark riesce (in molti casi) a essere migliore di Hadoop e del suo paradigma di calcolo MapReduce; vanta infatti dei tempi di calcolo fino a cento volte inferiori. L'utilizzo di Spark è più semplice di Hadoop; ciò nonostante tale semplicità e trasparenza portano l'utente a trascurare il suo funzionamento e questo comporta problemi per le sue performance. Spark è un oggetto complesso e al fine di poterne studiare le performance si è reso necessario uno studio sul suo funzionamento (non è possibile migliorare qualcosa che non si comprende).

L'oggetto di questa tesi è Spark ed in particolare lo studio delle sue prestazioni tramite le sopracitate tecniche di analisi di *Data Warehousing*. In particolare, viene proposta una metodologia per analizzare in maniera sistematica l'esecuzione di una o più applicazioni su Spark indipendentemente dal dominio applicativo. Sono stati inoltre effettuati dei test strutturati e pensati per poter comprendere gli aspetti critici da considerare quando si riscontrano problemi di inefficienza; i fattori considerati sono l'utilizzo della CPU e della memoria e, più in generale, la quantità di risorse da allocare per una esecuzione più efficiente delle applicazioni. Grazie a questa tecnica si ha la possibilità di esplorare le metriche relative allo svolgimento dei programmi mettendo l'utente nelle condizioni di individuare facilmente eventuali colli di bottiglia. Infine, tramite i vari test, sono state formulate delle best practices il cui scopo è quello di guidare nell'allocazione ottimale delle risorse.

La seguente tesi è così strutturata:

- nel primo capitolo viene introdotto Hadoop e il suo ecosistema;

- nel secondo capitolo viene trattato Spark dando particolare enfasi alla sua architettura e sul come un'applicazione viene eseguita;
- nel terzo capitolo si presente un data warehouse per lo studio delle performance;
- nel quarto capitolo viene descritta la fase di test e le relative conclusioni.

Desidero ringraziare mio padre,
il cui sostegno è stato fondamentale
Un ringraziamento speciale
anche a Francesco e Alessandro,
grazie a loro ho avuto la possibilità
di studiare Hadoop e Spark

Capitolo 1

Hadoop

In questo capitolo si presenta Hadoop e le sue componenti fondamentali. Spark e Hadoop forniscono gli strumenti per elaborare i big data, tuttavia non hanno funzioni identiche né sono mutualmente esclusivi poiché possono lavorare assieme. In alcune situazioni Spark è circa cento volte più veloce rispetto ad Hadoop ma ha la pecca di non disporre di un proprio storage distribuito. Per questo motivo i progetti nel settore dei big data prevedono la presenza di Spark e Hadoop contemporaneamente, affinché il primo possa usare il file system distribuito fornito dal secondo[9].

Il progetto Hadoop, sviluppato da Apache[1], è un *framework*[2] open-source per il calcolo parallelo e distribuito. Permette alle applicazioni di lavorare con migliaia di nodi e processare petabyte di dati. E' stato ispirato dall'algoritmo MapReduce di Google e dal Google File System (GFS). Alcuni tra i suoi utilizzatori più importanti sono: Ebay, Facebook, IBM e Twitter.

Le caratteristiche più importanti di Hadoop sono:

- scalabilità ovvero la sua capacità di crescere o diminuire di scala in funzione delle necessità e delle disponibilità;

[1]L'Apache Software Foundation (ASF) è una fondazione no-profit ed una comunità di sviluppo di progetti software come il web server Apache (il progetto principale) e la suite da ufficio Apache OpenOffice (gestita prima da Sun Microsystems e poi da Oracle America).

[2]Un framework è una infrastruttura generale che permette di semplificare lo sviluppo software lasciando al programmatore il contenuto vero e proprio dell'applicazione

- costi ridotti per il mantenimento dei dati;
- flessibilità nel trattare i dati ovvero la possibilità di processare dati strutturati e non;
- velocità di calcolo;
- *fault tollerance* ovvero la capacità di un sistema di non subire avarie (cioè interruzioni di servizio) anche in presenza di guasti.

Hadoop non è un singolo software ma si compone invece di un insieme di progetti. Tra i più importanti si ricordano HDFS, Yarn e MapReduce.

La fig. 2.4 mostra l'ecosistema di Hadoop ovvero l'insieme dei moduli di cui si compone.

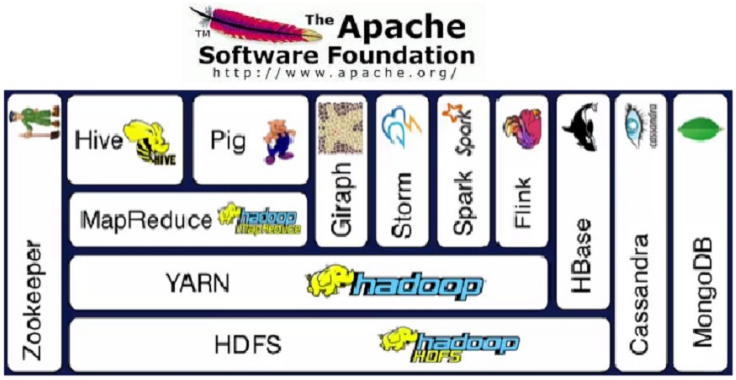

Figura 1.1: Ecosistema di Hadoop

Nel prosieguo del capitolo vengono trattati singolarmente i vari componenti, in modo da avere una panoramica sulle funzionalità e gli scopi dei singoli progetti.

1.1 MapReduce

MapReduce è un *framework* per scrivere applicazioni che processano grosse quantità di dati in parallelo su grandi cluster in maniera *fault-tollerant* e

su hardware d'uso comune; segue il principio del *divide et impera*, dividendo l'operazione di calcolo in più parti processate in modo autonomo.

Prima di proseguire si presentano alcune definizioni utili:

Definizione 1.1 (Map)**.** è il nome di una funzione di ordine superiore che applica una certa funzione ad ogni elemento di una lista.

Definizione 1.2 (Reduce)**.** è il nome di una funzione di ordine superiore che analizza una struttura dati e la riorganizza per mezzo di una funzione passata come argomento.

Definizione 1.3 (Job)**.** è l'intero processo di esecuzione: i dati in input, l'esecuzione del mapper e reducer e i dati in output.

Definizione 1.4 (Task)**.** Ogni Job è diviso in mapper e reducer; un task è la porzione di Job attribuibile ad ogni mapper e reducer.

Definizione 1.5 (Partizione)**.** E' l'insieme delle coppie <chiave, valore> che sono inviate ad un reducer.

Un'applicazione MapReduce divide il dataset in blocchi indipendenti che vengono processati da entità (task) in modo indipendente e parallelo. E' lo stesso *framework* che si occupa di organizzare i task, monitorarli e re-eseguirli nel caso in cui fallissero.

MapReduce opera in due fasi una di Map e una di Reduce. Durante il Map il dataset viene letto e convertito in un nuovo insieme di dati composto da tuple del tipo <chiave, valore>; durante il Reduce il dataset risultante dal Map viene ordinato e poi processato aggregando le tuple in insiemi più piccoli.

Per comprendere meglio il funzionamento si presenta un esempio. Si assuma di avere cinque file contenenti due colonne (una chiave ed un valore) che rappresentano una città e la corrispondente temperatura registrata. L'obiettivo è di individuare la temperatura massima di ogni città. Le informazioni di ogni città sono sparse sui vari file, non è quindi presente alcun tipo di ordinamento. Usando il *framework* MapReduce vengono assegnati cinque *map task*, uno per file, il cui output sarà la temperatura massima di ogni città.

Il risultato ottenuto in questa fase è parziale, ovvero gli output della fase di map contengono la temperatura più alta rispetto alle misurazioni contenute nel file analizzato da ogni task ma non rispetto all'insieme dei file. Tutti e cinque gli output verranno inviati ai *task reduce*, i quali combineranno l'input in un unico blocco e individueranno il valore più grande di ogni città.

La fig. 1.2 sintetizza quanto appena detto. Si nota che il *framework* legge il file di input da HDFS, lo divide in un insieme di partizioni e ognuna di esse viene processata prima da una fase di Map e (dopo un riordino delle tuple) da una fase di Reduce; infine il risultato viene scritto su disco.

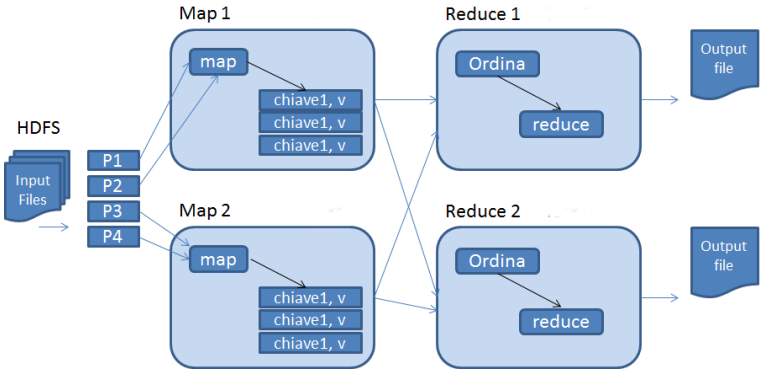

Figura 1.2: Funzionamento algoritmo MapReduce

1.2 HDFS

Hadoop Distributed File System (HDFS) è un *file system* distribuito ideato per soddisfare i requisiti di affidabilità e scalabilità. I file sono organizzati in blocchi di dimensione uguale (generalmente 64 o 128 MB) e vengono replicati in modo da garantire la propietà *fault tollerant* e rendere più efficiente il recupero dei dati. In altre parole, la quantità minima che HDFS può leggere o scrivere è chiamata blocco.

Il motivo per cui i file sono divisi in blocchi aventi stessa dimensione è l'efficienza: HDFS è stato pensato per l'archiviazione di grandi quantità di dati, poichè se si utilizzassero blocchi di dimensioni piccole si avrebbero una quantità di richieste di lettura e scrittura ingestibili.

HDFS ha un'architettura di tipo master-slave, infatti questa consiste di un singolo *NameNode* (il master) che gestisce il *namespace*[3] e regola l'accesso ai file da parte dei client; è responsabile anche di mantenere traccia delle locazioni di files e directory. Il *NameNode* salva periodicamente le modifiche del *file system* in un file di log (*edits*). Ogni volta che il *NameNode* viene avviato legge lo stato da un file chiamato *fsimage* e vi aggiunge le modifiche contenute nel file di log. Dato che questa operazione viene fatta solo all'avvio, il file *edits* può diventare importante. Per questo motivo è stato introdotto un servizio chiamato *Secondary NameNode* che periodicamente si occupa di unire i due. In aggiunta vi sono un insieme di *DataNode* che gestiscono lo storage, in genere uno per nodo, e rispondono alle richieste di lettura-scrittura da parte dei client gestendo quindi la creazione e cancellazione dei blocchi e la loro replica.

La fig. 1.3 descrive la struttura di HDFS. Questa si compone di un *NameNode* e di un insieme di *DataNode* contenenti i blocchi che a loro volta compongono un file.

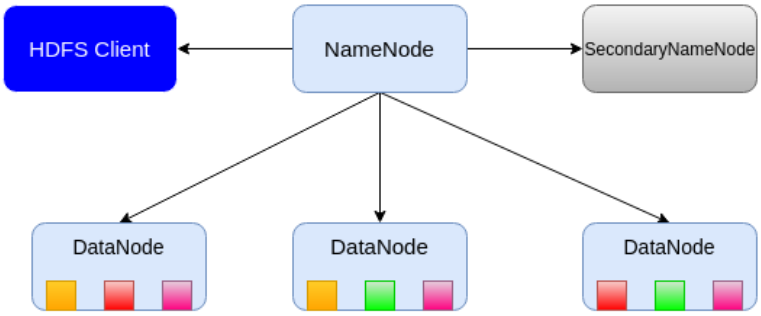

Figura 1.3: Architettura di HDFS

1.3 YARN

Apache YARN (Yet Another Resource Negotiator) è il gestore delle risorse di Hadoop. E' stato introdotto per migliorare il MapReduce ma supporta

[3]Con il termine namespace si intende la struttura ad albero e gerarchica del file system

anche altri paradigmi di calcolo parallelo. Rispetto alla prima versione è un sistema molto più generico infatti è stato riscritto come applicazione YARN con alcune sostanziali differenze. L'idea di base di YARN è di ottimizzare l'uso delle risorse disponibili in un cluster (CPU e memoria) e di pianificare le operazioni di calcolo quando più applicazioni ne fanno richiesta; in sostanza è stato disegnato per permettere a più applicazioni di condividere le risorse disponibili.

La struttura di Yarn è semplice, questa si compone di:

- Resource Manager: è l'agente che si occupa di gestire le risorse quando più applicazioni sono eseguite nel cluster;
- Node Manager: è il responsabile di un singolo nodo nel cluster e ha i compiti di monitorare l'utilizzo delle risorse e riportare le informazioni al Resource Manager;
- Container: è il diritto di un applicazione all'uso di una certa quantità di risorse su di uno specifico nodo (Node Manager).

Le applicazioni YARN hanno la seguente struttura di funzionamento:

1. l'utente scrive la sua applicazione che prende il nome di "Application Master" e tramite il client YARN richiede al resource manager la registrazione al cluster;
2. il resource manager richiede ad ogni nodo la creazione di un container per eseguire l'application master ;
3. l'application master viene quindi mandato in esecuzione sul container;
4. una volta in esecuzione il master si moltiplica, richiedendo a sua volta al resource manager altri container;
5. il resource manager si occupa di gestire le richieste dei container da parte dei vari application master in esecuzione sul cluster. L'allocazione è dinamica, cioè ogni application master si troverà ad avere a disposizione più o meno container a seconda delle disponibilità del cluster;

6. il client a questo punto comunica direttamente con il suo Application master per avere i risultati dell'esecuzione dell'applicazione.

La fig. 1.4 rappresenta Yarn e il suo funzionamento: vi sono un certo numero di Node Manager nei quali vengono allocati uno o più container relativi ad una o più applicazioni; ogni container corrisponde ad una certa quantità di risorse per l'esecuzione dell'applicazione.

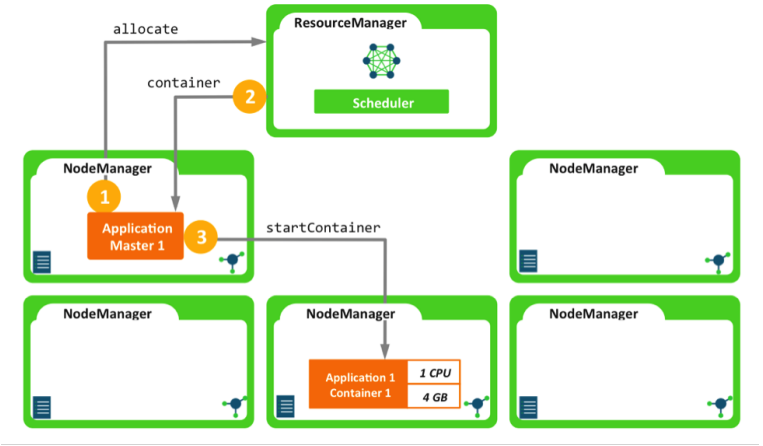

Figura 1.4: Architettura di Yarn

Capitolo 2

Spark

2.1 Apache Spark

Spark è una piattaforma di computazione su cluster designata per essere veloce, general-purpose ed altamente accessibile. Dal punto di vista della velocità, Spark estende il popolare modello MapReduce supportando in modo efficiente altri tipi di funzioni. È possibile infatti utilizzare queries interattive ed elaborare i dati in streaming, cioè in tempo reale. Quando è necessario processare grandi insiemi di dati, la velocità di computazione diventa fondamentale: essa determina le prestazioni permettendo di ottenere i risultati in pochi secondi piuttosto che dopo diversi minuti o anche ore. Una delle principali caratteristiche di Spark, che migliora la velocità di computazione, è la possibilità di eseguire calcoli in memoria. Spark inoltre, come in MapReduce, permette di eseguire applicazioni direttamente sul disco, ma con prestazioni più efficienti. Alla base dell'architettura di Spark è presente HDFS, tuttavia Spark è in grado di lavorare anche con il file system locale o l'*Amazon S3*[4]. Sopra il livello del file system si incontra il *cluster manager*. Anche qui Spark è molto flessibile in quanto è stato creato per potersi integrare con tre diversi manager ovvero lo Standalone Scheduler (nativo di Spark), Yarn e Mesos.

Dal punto di vista dell'accessibilità, Spark mette a disposizione APIs per svariati linguaggi quali Python, Java, Scala ed R. Sono inoltre incluse all'interno ricche librerie (SQL e DataFrames, Spark Streaming, MLlib, GraphX)

che semplificano il lavoro dell'utente. Il motore di Spark, quindi, è in grado di elaborare diversi carichi di lavoro e permette di combinare tali carichi all'interno di un unico processo di elaborazione. Prima di Spark, erano necessari diversi sistemi distribuiti per elaborare i diversi carichi di lavoro.

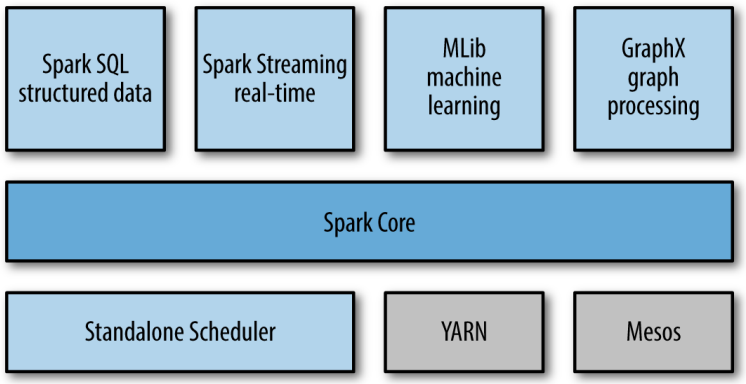

Figura 2.1: Le componenti ad alto livello di Spark

2.2 Differenze con MapReduce

Spark è stato progettato e implementato da un gruppo di ricercatori dell'università di Berkeley nel 2010. In sostanza, costoro si sono chiesti come potevano migliorare le performance dei sistemi distribuiti MapReduce, dato che:

- Map-Reduce è lento, perchè ad ogni job legge i dati da HDFS, carica da disco, mappa, riduce, scrive i dati su disco;
- Map-Reduce è poco adatto a fare elaborazioni complesse e ripetute sullo stesso dataset, perchè in una sequenza di job c'è una lettura e una scrittura su disco;
- Map-Reduce non è adatto a operazioni di tipo iterativo [1].

[1]Sono operazioni formate da cicli di analisi e aggiornamenti consecutivi sullo stesso dataset, tipici quando ad ogni aggiornamento si deve riprendere in mano il precedente set di dati elaborati e aggiornarlo di nuovo

Hanno quindi pensato di progettare un nuovo sistema di elaborazione distribuita che consentisse di specificare sequenze complete di operazioni, anche iterative, in un solo job, riducendo la complessità del codice dell'applicazione di elaborazione. Inoltre questo permette di mantenere i dati in memoria fino al termine dell'intera sequenza di elaborazione, evitando di leggere e scrivere su disco (a meno che non sia lo stesso sviluppatore a richiederlo) e garantisce la stessa capacità di Hadoop di resistere alla perdita di nodi del cluster e quindi perdita di parte dei dati. Hanno quindi sviluppato il concetto di Resilient Distributed Dataset (RDD) che è la teoria alla base del sistema Spark.

2.3 Resilient Distributed Dataset

Un RDD in Spark è semplicemente una collezione distribuita ed immutabile di oggetti[7]. E' in sostanza un set di dati suddiviso in partizioni (un file spezzato in tanti segmenti) che ha alcune proprietà chiave per il suo funzionamento:

- Ogni RDD è immutabile, cioè una volta creato non lo si può cambiare, se non creandone un altro mediante una trasformazione;
- Ogni RDD può solo essere creato inizialmente a partire dai dati su disco (presi da HDFS) oppure a partire da altri RDD; questo approccio serve perchè un singolo pezzo (partizione) dell'RDD possa essere ricostruito a partire dalla sequenza di trasformazioni che lo hanno generato;
- Ogni RDD è descritto da un set completo di metadati che consentono la ricostruzione delle sue partizioni in caso di fault; grazie a questi metadati è possbile conoscere dove si trovano le partizioni, quali sono gli RDD padre, quale è la sequenza di trasformazioni detta lineage graph, che hanno generato l'RDD.

Spark nasce come un sistema per creare e gestire job di analisi basati su trasformazioni di RDD. Dato che questi nascono e vivono in memoria,

l'esecuzione di lavori iterativi o che trasformano più volte un set di dati, sono immensamente più rapide di una sequanza MapReduce (10, anche 100 volte più veloci, perchè il disco non viene mai, o quasi, impiegato nell'elaborazione).

Esistono due tipologie di operazioni su un RDD: trasformazioni e azioni. Le trasformazioni creano un RDD a partire da uno esistente mentre le azioni restituiscono un risultato all'utente dopo aver effettuato la computazione.

Un concetto molto importante che riguarda l'esecuzioni delle trasformazioni è la lazy evaluation. Per l'appunto, le trasformazioni sono valutate in modo pigro, cioè esse non saranno effettuate finché non si presenterà un'azione. Per lazy evaluation si intende che quando è richiamata una trasformazione su un RDD, l'operazione non è immediatamente eseguita, ma sarà eseguita solo quando sullo stesso RDD sarà richiamata un'azione.

A causa della lazy evaluation ogni qual volta viene eseguita un'azione, Spark dovrebbe ricomputare l'RDD insieme a tutte le sue dipendenze. Questo comporterebbe un grande spreco nel caso in cui fossero eseguite più azioni sullo stesso RDD. Nel caso in cui si dovesse usare lo stesso RDD più volte, è possibile utilizzare il metodo persist che permette di mantenere in memoria l'RDD senza doverlo ricalcolare. Esistono diversi livelli di storage con cui è possibile richiamare il metodo persist:

- MEMORY ONLY - Memorizza l'RDD in memoria, riducendo così il tempo di latenza dovuto alla lettura dei dati. Nel caso in cui non ci sia abbastanza spazio in memoria per memorizzare l'RDD, quest'ultimo non sarà memorizzato e sarà ricomputato nel caso sarà richiamata un'azione su esso. In tale modalità, i dati sono salvati in memoria senza essere precedentemente serializzati;

- MEMORY ONLY SER - È del tutto analogo al precedente livello di memorizzazione. L'unica differenza è che esso, al contrario del precedente, serializza i dati prima di memorizzarli in memoria. Questo processo da una parte comporta una diminuzione dello spazio in memoria utilizzato, dall'altra parte comporta un maggior overhead, in termini di prestazioni, dovuto ai processi di serializzazione e deserializzazione dei dati;

- MEMORY AND DISK - Simile al MEMORY ONLY, con la differenza che nel caso in cui i dati non dovessero essere contenuti totalmente in memoria, Spark sposta le partizioni dei dati che non entrano in memoria sul disco. In tal caso, lo spazio utilizzato è alto poiché non viene adoperato il processo di serializzazione, in cambio i tempi delle prestazioni sono medi;
- MEMORY AND DISK SER - Analogo al MEMORY ONLY SER, l'unica variante consiste nel memorizzare le partizioni dei dati che non rientrano in memoria sul disco. Dato che i dati sono serializzati, lo spazio utilizzato è basso, ma i tempi delle prestazioni sono alti;
- DISK ONLY - Memorizza gli RDD solamente sul disco. Questo non comporta un grande spreco di spazio, poiché i dati sono memorizzati sul disco, e proprio per tale motivo invece i tempi delle prestazioni sono alti.

La fig. 2.2 rappresenta un esempio di trasformazione Map in cui a partire da un RDD di stringhe si ottiene un nuovo RDD formato dalle coppie (String, Intero). Questa è ottenuta applicando la funzione Word => (Word, 1) ad ogni riga dell'RDD.

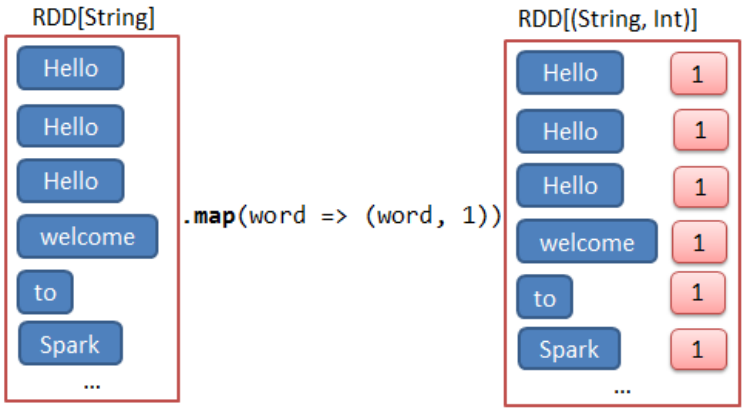

Figura 2.2: Esempio di trasformazione

Un'altra caratteristica di Spark è la fault tollerance. Con questo termine si indica la capacità del sistema di non perdere i dati nel caso in cui si doves-

serò verificare problemi (perdita di un nodo ad esempio). Questa capacità è insita nel concetto di RDD, infatti prima di eseguire qualsiasi operazione Spark analizza il codice dell'applicazione e ne crea un piano logico (Directed Acylcic Graph)[2] ovvero un insieme di trasformazioni e azioni che verranno applicate sui dati. Se un RDD o una sua partizione dovessero essere persi Spark è in grado di ricostruirla. La fig. 2.3 è un esempio di DAG e mostra due RDD iniziali A e C; questi subiscono alcune trasformazioni (map, filter, groupBy) permettendo così di ottenere nuovi RDD e infine il risultato, ovvero F. Nell'eventualità in cui una delle partizioni dovesse esser persa Spark conosce le operazioni necessarie per poterla calcolare nuovamente.

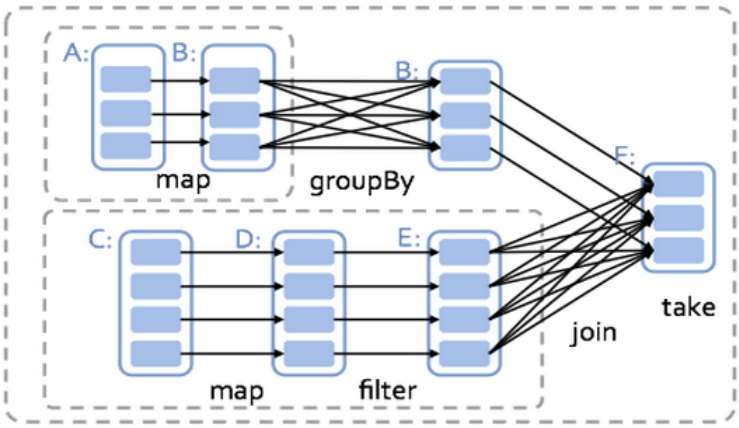

Figura 2.3: Esempio di Lineage Graph

Il funzionamento complessivo di Spark avviene in questo modo: lo sviluppatore scrive la propria applicazione definendo le trasformazioni e calcoli che desidera fare sul set di dati. Ad esempio una elaborazione su file di log in cui l'obiettivo è contare quanti errori si sono verificati in un determinato arco di tempo. Nel programma possono esserci trasformazioni (generano un RDD) o azioni (fanno calcoli sull'RDD e restituiscono il risultato), organizzate in sequenze o anche in cicli iterativi. Nel caso dei log ci saranno mappature (cioè caricamenti in array), filtraggi (selezione solo dei log di errore e poi di

[2]Un DAG è un particolare tipo di grafo diretto che non ha cicli e in cui ogni nodo è una partizione di un RDD e ogni vertice è una trasformazione

quelli in un certo intervallo di tempo) e un'azione di conteggio sul dataset finale a valle del filtraggio. Il programma viene passato allo scheduler di Spark che sulla base del codice costruisce il grafo delle trasformazioni degli RDD che sono richieste. In base al grafo ottenuto lo scheduler determina il miglior modo possibile per distribuire i lavori di trasformazione sui nodi. Questo deve tenere conto di due possibili tipi di dipendenze tra RDD consecutivi: dipendenza "stretta" (narrow) in cui il figlio dipende direttamente dal padre e dipendenza "ampia" (wide) dove il figlio è una combinazione di più RDD padre. La fig. 2.4 permette di comprendere meglio le due tipologie di dipendenza

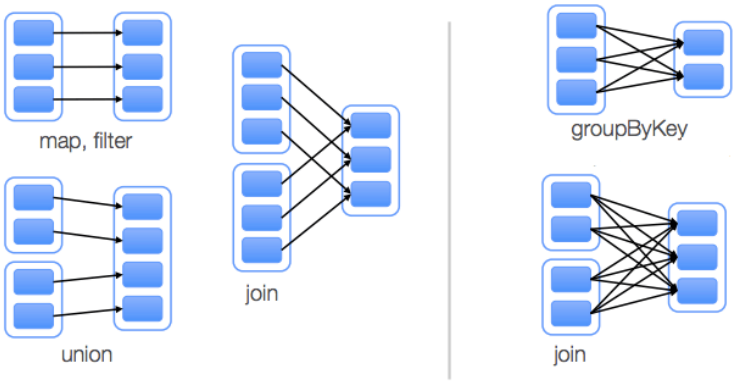

Figura 2.4: Dipendenze tra RDD. A sinistra si ha una dipendenza diretta mentre a destra composita

Se la dipendenza tra RDD è di tipo wide, Spark deve riorganizzare i dati per poter continuare l'elaborazione. Si immagini di dover contare il numero di volte in cui una parola è contenuta in un file (distribuito su più nodi). Ogni nodo conta il numero di occorrenze per la porzione del data set che gli è stato assegnato ma una volta che tutti i nodi hanno ottenuto il risultato parziale è necessario unirli per poter avere il conteggio finale. Unire vuol dire trasferire i dati tra una macchina e un'altra. Questa fase prende il nome di Shuffle ed è una delle operazioni più costose su Spark in quanto comporta l'utilizzo della rete.

2.4 Architettura

Spark utilizza un'architettura di tipo master-slave con un coordinatore centrale e molti worker. Il coordinatore centrale è detto driver e comunica con un numero potenzialmente grande di worker che prendono il nome di executor. Sia il driver che gli executor sono dei processi Java separati l'uno dall'altro e l'insieme dei due forma un'applicazione.[8] Per meglio comprendere il funzionamento si riporta il ciclo di vita di un applicazione:

1. L'utente lancia un'applicazione usando il comando spark-submit;
2. Il comando spark-submit lancia il driver program che invoca il metodo main();
3. Il driver program chiede al cluster manager di allocare le risorse per poter lanciare gli executor;
4. Il cluster manager lancia gli executor a nome del driver program richiedente;
5. Il driver process viene eseguito tramite il programma utente: quando vengono richieste trasformazioni o azioni sugli RDD il driver manda agli executor, tramite il cluster manager, dei task da eseguire;
6. I task vengono eseguiti dagli executor che generano i risultati;
7. Non appena termina il main() o viene invocato il metodo stop() sullo SparkContext, il driver chiede la disallocazione delle risorse al cluster manager e la terminazione degli executor.

Un'applicazione è lanciata su un cluster utilizzando un servizio chiamato cluster manager (vedi Yarn).

Il Driver è il processo dove il main() dell'applicazione viene lanciato, che crea lo SparkContext[3], gli RDD ed effettua le trasformazioni e le azioni. Quando il Driver viene lanciato ha due compiti:

[3]Lo SparkContext è il primo oggetto creato da un programma Spark. Questo fornisce le informazioni per poter accedere al cluster.

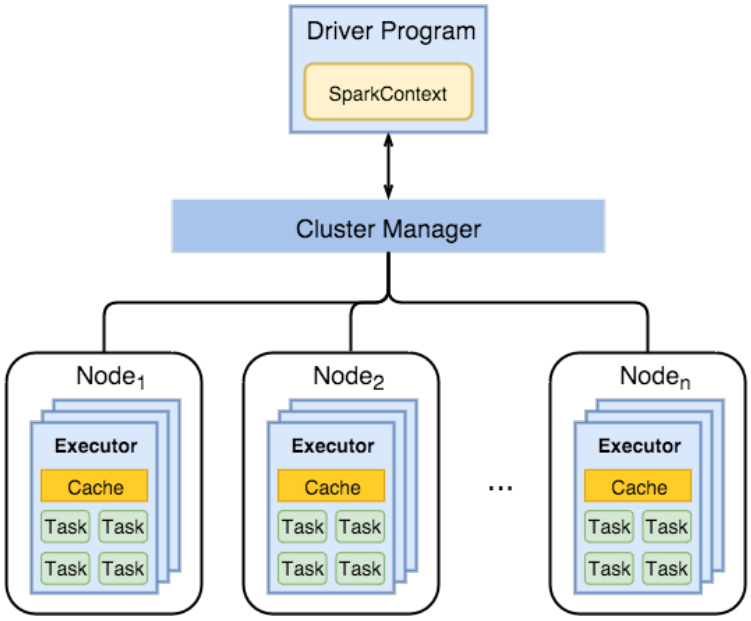

Figura 2.5: Schema dell'architettura Spark

- converte il programma in un insieme di task: parte dalla creazione del DAG e lo trasforma in un piano fisico di esecuzione cercando di ottimizzare le operazioni da svolgere. In particolare genera degli Stage ognuno dei quali si compone di un insieme di task da inviare agli executor. I task sono l'unità fondamentale di lavoro esistente in Spark;

- organizza i task da inviare agli executor: quando questi iniziano la loro vita si "registrano" al driver, in questo modo esso ha una visione completa dell'applicazione in ogni istante. Il driver controlla gli executor disponibili e invia loro i task seguendo come criterio quello del data locality[4].

Infine il Driver permette di seguire la vita dell'applicazione tramite una interfaccia web.

Indipendentemente dal cluster manager scelto Spark fornisce un singolo script per il lancio delle applicazioni chiamato spark-submit. Attraverso l'u-

[4]Con il termine data locality si riferisce alla prossimità dei dati rispetto a dove il task viene eseguito. Generalmente sarà il task ad essere inviato dove il dato risiede in modo da diminuire la banda usata.

tilizzo di diverse opzioni lo spark-submit può connettersi a differenti cluster manager e controllare quante risorse l'applicazione chiede (verrà approfondito nell'ultimo capitolo).

Gli executor sono dei processi Java che si occupano di eseguire i task assegnati dal driver. In realtà i compiti degli executor sono due, il primo è quello appena descritto, cioè eseguire i task che li vengono assegnati e ritornare il risultato utile per l'applicazione; il secondo è quello di fornire meccanismi di salvataggio in memoria per gli RDD che sono stati messi in cache dal programma utente attraverso un servizio, dell'esecutore stesso, che si chiama Block Manager. Il ciclo di vita degli executor è strettamente connesso a quello dell'applicazione, mentre non è vero il contrario, nel senso che gli executor sono lanciati una sola volta ad inizio applicazione e normalmente terminano quando termina l'applicazione; inoltre se avviene un problema in un executor, esso termina ma l'applicazione continua la sua esecuzione. La memoria per il salvataggio degli RDD è quindi messa a disposizione dagli executor; come detto, questo è uno dei punti di forza del sistema Spark e per questo si propone un approfondimento.

2.4.1 Gestione della memoria

A partire dalla version 1.6.0+ il modello della memoria è stato modificato[5]. Il vecchio modello è chiamato Legacy ed è disabilitato di default (tuttavia si può modificare usando il parametro *spark.memory.useLegacyMode*). Il *Memory Manager*[5] in Spark è il responsabile della *Heap Memory* (sito della memoria della Java Virtual Machine nella quale vengono allocati gli oggetti appena creati) di un Executor. Questo gestisce la frazione di memoria dedicata allo storage (caching) e quella dedicata allo shuffle (quindi tutte le operazioni quali join, sort, aggregazioni, ecc.).

In fig. 2.6 vi è un dettaglio di come la memoria viene ripartita. Il totale prende il nome di Java Heap ed è divisa in tre macrosezioni. Nello specifico si nota:

[5]Con Memory Manager si intende il processo di allocazione di nuovi oggetti e la rimozione di quelli non usati dalla memoria

- Reserved Memory: è la memoria riservata al sistema (un minimo di 300 MB) e non partecipa al processo di calcolo. La sua dimensione può essere modificata anche se non è consigliato farlo; infatti se non si assegna un minimo di $1.5 \times$ *Reserved Memory* (pari a 450 MB) l'esecuzione fallisce;

- User Memory: è la porzione di memoria dedicata all'utente; questa comprende le strutture dati per le trasformazioni. La sua dimensione è calcolabile come:

$$(JavaHeap - ReservedMemory) \times (1.0 - spark.memory.fraction)$$

  dove *spark.memory.fraction* è di default pari a 0.75.

- Spark Memory: è la parte di memoria gestita da Spark ed è calcolabile come:

$$JavaHeap - (ReservedMemory \times spark.memory.fraction)$$

  Ad esempio con 4 GB di Heap la parte dedicata a Spark è 2847 MB. Questa è divisa in due parti, *Storage Memory* e *Execution Memory*. La prima è usata per salvare i dati in cache e per la deserializzazione, mentre l'*Execution Memory* è l'ubicazione di memoria in cui gli oggetti vengono salvati durante la fase di Shuffle. Se non vi è abbastanza spazio Spark scrive su disco.

La versione *Legacy* del Memory Manager ha due svantaggi, ovvero il sotto utilizzo della Heap Memory e l'intervento manuale per ottimizzarla. Nello specifico si ha che:

- la quantità di RAM dedicata all'execution e allo storage sono statiche in quanto determinate dall'utente che manualmente imposta il parametro *spark.storage.memoryFraction*;

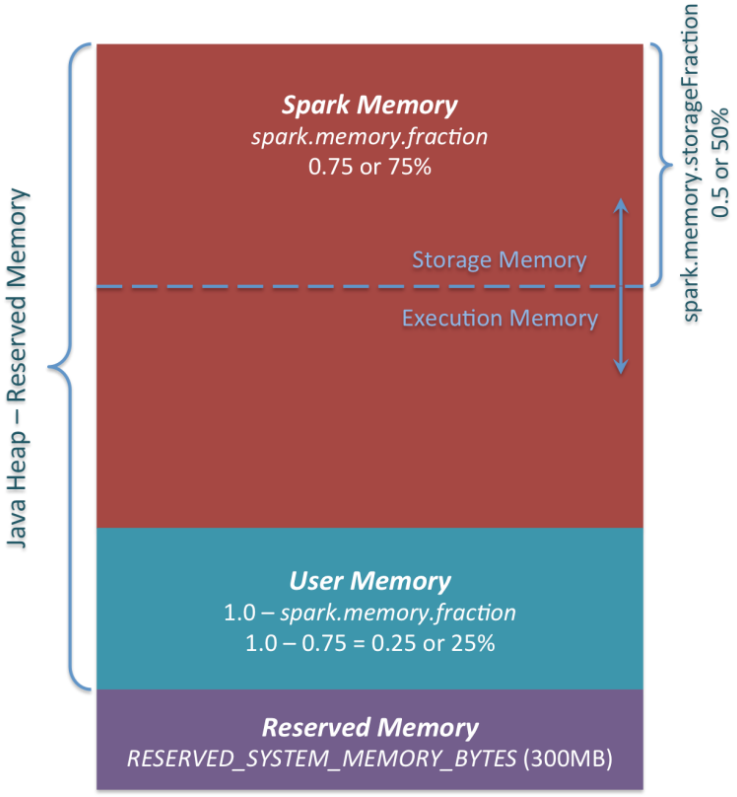

Figura 2.6: Struttura della memoria

- l'utente deve stimare la memoria di storage e quella di shuffle; questo deve essere fatto per ogni esecuzione.

Nella versione *Unified* si è creata una zona "cuscinetto" (sia M) tra le due memorie che si contendono la *spark.memory.fraction*. La novità è di permettere alla Storage e alla Execution di prestarsi memoria in base alle necessità. A causa della natura della Execution Memory questa non può essere forzata nel rilasciare blocchi, in quanto i dati sono utilizzati in calcoli intermedi e complessi che non possono essere bloccati. Questo non è vero per la Storage Memory; infatti se un blocco dati viene spostato su disco Spark aggiorna i metadati riflettendo la loro nuova posizione (o cancellazione).

In sostanza si può forzare un blocco dalla Storage Memory ma non dalla Execution. Quando l'Execution Memory può chiedere memoria dalla Storage? Semplicemente quando la Storage non utilizza tutta la sua memoria disponibile. Al contrario, la Storage Memory può prendere memoria in prestito solo se vi è spazio disponibile nella Execution. In dettaglio esiste una sotto regione di M (sia R) dove i blocchi dentro l'area di Storage non vengono mai rilasciati. Quindi la memoria di Storage può prestare memoria ma conserverà sempre una quantità R di Ram per sé. Questo è dovuto al modo in cui il sistema Spark è stato pensato; in genere i calcoli che avvengono nell'area della execution memory sono temporanei e la memoria viene rilasciata, mentre il salvataggio in memoria dei blocchi sarà utilizzato più volte nel corso dell'applicazione. Per questo motivo esiste una zona R che non può essere modificata.

Un'altro aspetto interessante introdotto nella versione *Unified* è la gestione della memoria da parte dei task. L'assegnamento della memoria è dinamico. Con questo si intende che ogni task può prendere una porzione di memoria variabile in base al numero di questi. Ad esempio se vi sono quattro task che vengono eseguiti in parallelo allora ognuno avrà a disposizione un quarto della memoria disponibile. In generale se N è il numero di task ognuno avrà a disposizione una percentuale pari a 1/N.

2.4.2 Job

Un Job è l'elemento più in alto nella gerarchia di esecuzione di Spark. Ogni Job corrisponde ad un'azione chiamata dall'applicazione. Un modo per concepire un'azione è di immaginarla come una funzione che materializza i dati fuori da un RDD in un qualche sistema di storage esterno. Quindi un'azione restituisce un risultato al Driver o lo scrive sul file system HDFS.

Dato che i vertici del grafo di esecuzione di Spark si basano su dipendenze tra le trasformazioni degli RDD, un operazione che restituisce qualcosa di diverso da un RDD non può avere figli; quindi un insieme arbitrario di trasformazioni può essere associato ad un grafo di esecuzione. Ogni volta che un azione si presenta Spark deve includere, nel piano d'esecuzione, tutte le trasformazioni necessarie per valutare l'RDD finale che ha causato l'azione.

La fig. 2.7 illustra quanto appena appena detto: un'azione comporta un Job e questo a sua volta richiede un insieme di trasformazioni che provocano degli shuffle.

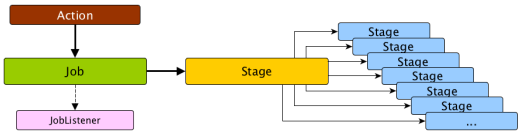

Figura 2.7: Schema di esecuzione di un azione

2.4.3 Shuffle

Lo Shuffle è uno degli aspetti più interessanti dell'infrastruttura di Spark. Per comprendere cos'è lo Shuffle e quando è inevitabile si presenta un esempio. Si immagini una lista contenente delle chiamate e di voler calcolare quante ne siano state fatte quotidianamente. La prima cosa da fare in Spark è riconoscere il giorno come una chiave e associare ad ogni record il valore 1 (c'è stata almeno una chiamata); fatto ciò basta contare il numero di volte in cui compare la stessa chiamata e si ha la risposta. Tuttavia quando le informazioni sono salvate in maniera sparsa su un cluster il singolo nodo

non può sapere se vi sono chiavi ripetute su altri nodi. L'unico modo è di trasferire tutti i valori aventi stessa chiave sulla stessa macchina. L'evento di "trasferire" i dati è il processo di Shuffle ovvero una riorganizzazione dei dati che richiede l'utilizzo di banda di rete. Una definizione più rigorosa è la seguente: lo shuffle è il processo di redistribuzione dei dati attraverso le partizioni che può causare lo spostamento dei dati attraverso le JVM. Va precisato inoltre che lo shuffle divide i vari stage.

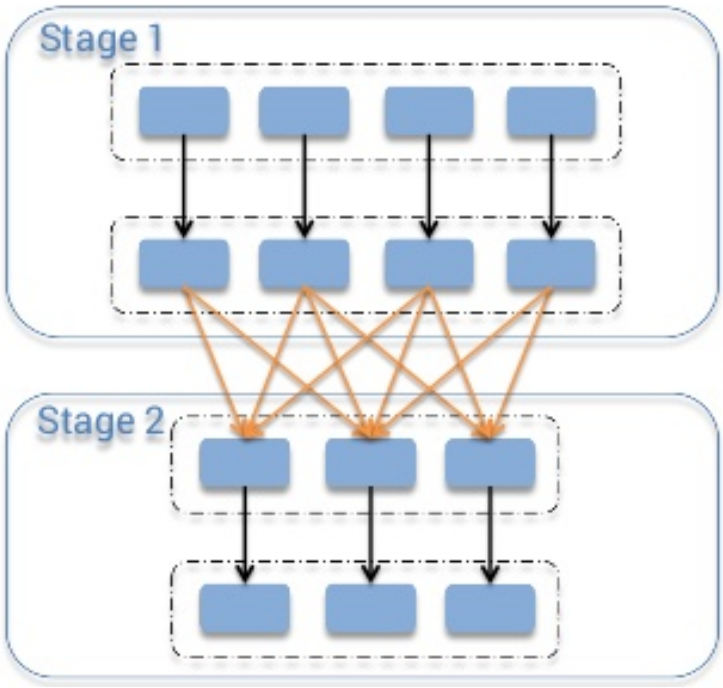

Figura 2.8: Operazione di shuffle

La fig. 2.8 illustra il concetto appena definito: i dati presenti nelle partizioni vengono riorganizzati in quanto questi risiedono su nodi differenti; ovviamente questo comporta l'utilizzo di banda di rete.

La fase di shuffle si divide in due parti: scrittura e lettura. La scrittura è un compito semplice se non è richiesto un output ordinato; ogni record dell'RDD in questa fase è partizionato e reso persistente ovvero mantenuto in memoria. La fig. 2.9 spiega meglio il funzionamento: vi sono quattro Shuffle Task (due per core) sullo stesso Executor. Il risultato di ogni task

viene diviso su un insieme di *bucket* (sono dei buffer[6]) che sono tanti quanti il numero di partizioni dell'RDD finale; successivamente i *bucket* vengono scritti su disco (per non perdere il risultato intermedio). Ogni record viene quindi inviato al bucket della sua corrispondente partizione e ordinato; questa rappresenta la fase di lettura o shuffle read.

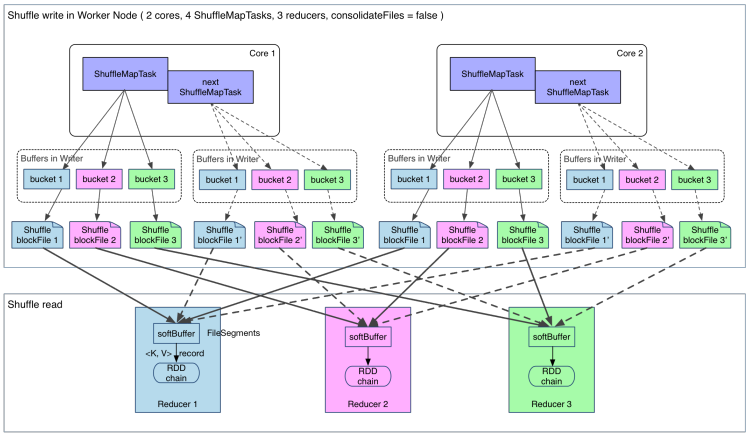

Figura 2.9: Operazione di shuffle

L'operazione di shuffle è quindi un operazione indispensabile e costosa in quanto:

- partiziona i dati: questo potrebbe significare un lavoro di ordinamento;
- serializza o deserializza i dati per permettere il trasferimento di questi attraverso la rete;
- comprime i dati;
- comporta frequenti operazioni di lettura e scrittura su disco.

[6]un buffer, termine della lingua inglese che significa letteralmente tampone, è una zona di memoria usata temporaneamente per l'entrata o l'uscita dei dati, oppure per velocizzare l'esecuzione di alcune operazioni. I buffer sono necessari per coprire i tempi di latenza dei collegamenti di rete. Due dispositivi che devono inviarsi grandi quantità di dati, in un sistema monodirezionale, dovranno salvare i dati da inviare in un buffer, per poi inviarlo quando il canale è libero.

2.4.4 Stage

Una stage è un insieme di task indipendenti (quindi possono lavorare in parallelo) aventi tutta la stessa funzione e in cui tutti i task hanno le stesse dipendenze. Ogni DAG di task lanciato dal DAGScheduler è diviso in stage ogni volta che si rende necessario uno shuffle. Ogni stage può essere uno *shuffle map stage*, nel caso in cui i risultati dei task sono un input per un altro stage, o un *result stage*, nel caso in cui i suoi task calcolino direttamente l'azione che ha avviato il job (ad esempio, count(), save() , eccetera). Ogni stage ha anche una jobId, che identifica il Job che lo ha lanciato.

2.4.5 Task

Un Task è l'unità fondamentale di esecuzione. Come già detto è un comando inviato dal Driver ad un Executor e ne esistono di due tipi: ShuffleMapTask e ResultTask. Il primo è un risultato intermedio il cui output sarà l'input del task successivo, mentre al contrario il secondo restituisce un risultato al driver. Indipendetemente dalla tipologia ogni task effettua tre operazioni: carica l'input, esegue il task e materializza l'output come shuffle o risultato per il driver. E' importante sapere che ogni task elabora una sola partizione. La fig. 2.10 mostra come un Executor esegue un insieme di task a seconda del numero di cores a disposizione.

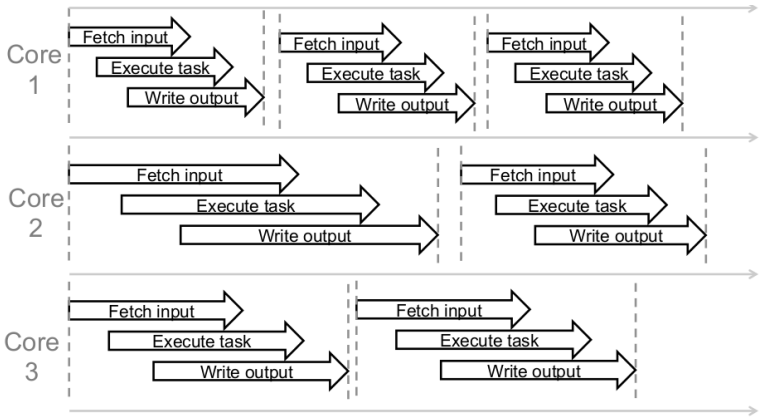

Figura 2.10: Executor e task

Capitolo 3

Data Warehouse per le performance

In questo capitolo verrà esposta una metodologia per analizzare e comprendere quanto accade durante la vita di un'applicazione Spark grazie all'utilizzo di un data warehouse. Spark fornisce un'interfaccia web (*Spark History Server*) per il monitoraggio delle applicazioni tuttavia questa è limitativa per lo scopo di questa tesi; pertanto si è proceduto alla riorganizzazione delle informazioni prodotte da Spark sotto forma di data warehouse. La prossima sezione di questo capitolo viene dedicata alla descrizione dello *Spark History Server* e succesivamente si presenta il data warehouse.

3.1 Spark History Server

Una delle componenti di Spark è lo SparkListener. Questa è una classe che intercetta gli eventi di un'applicazione quando vengono generati dagli scheduler durante l'esecuzione; detto in altre parole uno SparkListener riceve informazioni relative agli eventi Job, Stage e Task, al loro inizio e completamento ed anche informazioni su eventi infrastrutturali come un executor che viene aggiunto o rimosso oppure un RDD che viene spostato dalla memoria. Tutte queste informazioni possono essere consolidate in un file di log, specifico di ogni applicazione, per il debugging. Spark utilizza questo

"listener" per l'History Server che, come suggerisce il nome, è un interfaccia web che permette di monitorare sia le applicazioni in tempo reale che quelle già concluse. Questa interfaccia permette di controllare cosa accade durante l'esecuzione di un Job; in particolare è possibile visualizzare l'insieme degli stage che portano al completamento dello stesso sia in forma grafica che tabellare. Le informazioni che vengono mostrate per lo stage sono la sua durata, la dimensione dell'input e dell'output, e la quantità di dati che sono stati trasferiti durante la fase di shuffle. E' inoltre possibile controllare i task eseguiti all'interno di ogni stage e, per ognuno di essi, controllare le metriche come la durata totale, il tempo speso per il garbage collection o il tempo impiegato per la serializzazione. Inoltre anche gli RDD sono presi in considerazione ed è possbile conoscere la loro dimensione, la loro posizione in memoria, il numero di partizioni di cui si compongono ed il livello di storage.

Ogni volta che un'applicazione termina il suo ciclo di vita viene prodotto un file di log ed è grazie a questa fonte dati che lo Spark History Server ricostruisce la vita dell'applicazione. Il file di log è di tipo JSON[1] e ad ogni riga corrisponde un evento con una serie di informazioni collegate; gli eventi possono essere:

- Application Start/End: fornisce informazioni come il nome, l'identificativo dell'applicazione, il momento di lancio, l'utente che ha lanciato l'applicazione;
- JobStart/JobEnd: definisce l'inizio di un nuovo Job riportando informazioni sugli Stage che dovranno essere eseguiti, il numero di partizioni, il livello di storage e la dimensione in memoria degli RDD;
- StageSubmitted/StageCompleted: descrive l'inizio e la fine dello stage riportando i task eseguiti;
- TaskStart/TaskEnd: contiene un insieme di statistiche e metriche riguardanti ogni task, inoltre ogni task ha un proprio identificativo ed è possibile conoscere quale partizione ha processato;

[1]JSON, acronimo di JavaScript Object Notation, è un formato adatto all'interscambio di dati fra applicazioni client-server.

- Environment: contiene informazioni sull'ambiente di esecuzione. Tra le tante informazioni riportate le più importanti sono le impostazioni fornite al momento del lancio quali numero di cores, memoria assegnata ad ogni executor, ecc.

In fig. 3.1 è mostrato un esempio delle informazioni riportate dall'History Server relativamente agli executor; questi dati sono automaticamente aggregati ovvero sono il risultato della somma delle metriche dei task eseguiti in un particolare executor.

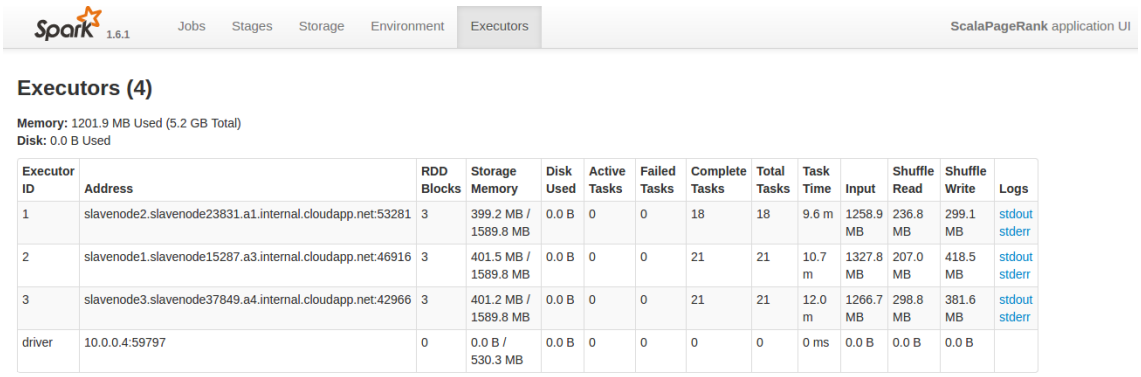

Figura 3.1: Esempio di informazioni riportate dall'History Server

Il file di log generato al termine di un'applicazione è utile per poter comprendere ciò che accade a basso livello e quindi valutarne le performance. Sebbene l'History Server rappresenti uno strumento potente per il debugging, i dati sono automaticamente aggregati dallo stesso; questo è un problema in quanto non è possibile esplorare le metriche generate durante l'esecuzione e il confronto tra più esecuzioni. In conclusione questo è un'ottimo strumento ma non permette di condurre studi approfonditi sulle performance di Spark, per questo motivo è stato progettato e implementato un data warehouse che verrà descritto nella prossima sezione.

3.2 Data Warehouse

Un data warehouse è una raccolta di dati organizzata per soggetti, integrata, non volatile, e variabile nel tempo, di supporto alle decisioni[3].

- Organizzata per soggetti: nei sistemi di supporto alle decisioni i dati sono organizzati per analizzare dei soggetti di interesse. Il focus si sposta dalle transazioni, caratteristica dei sistemi operazionali, all'analisi di una specifica applicazione o funzione aziendale;

- Integrata: la caratteristica distintiva principale di un datawarehouse è l'integrazione di dati provenienti da fonti eterogenee. L'integrazione avviene attraverso un processo di estrazione, trasformazione e caricamento dei dati dalle sorgenti al data warehouse;

- Non volatile: I dati sono caratterizzati da accessi in sola lettura e non sono soggetti a modifiche; per questo motivo mantengono la loro integrità nel tempo. Ciò comporta un notevole vantaggio anche in fase di progettazione in quanto non deve essere gestita la concorrenza tra l'aggiornamento dei dati e gli accessi in lettura;

- Variabile nel tempo: mentre i database operazionali conservano solo i dati più recenti, un data warehouse è pensato per fornire supporto alle decisioni strategiche, dunque necessita di un orizzonte temporale più ampio al fine di poter analizzare i cambiamenti nel tempo;

- Supporto alle decisioni: lo scopo principale è di supportare il processo decisionale inteso come capacità di rispondere rapidamente ai mutamenti.

Nel processo di data warehousing si parla di "fatto" in quanto questo rappresenta la misurazione di un fenomeno. Ogni fatto è caratterizzato da un insieme di attributi numerici, le misure, che lo descrivono. La tabella del fatto consiste di misure e chiavi esterne che referenziano delle chiavi primarie nelle dimensioni. Al contrario del fatto una dimensione è utilizzata per contestualizzare le misure dei fatti e per applicare filtri e raggruppamenti

di dati. Ad esempio la dimensione cliente potrebbe contenere degli attributi quali: nome, data di nascita, sesso etc. Molte dimensioni hanno una gerarchia di attributi che supportano il *drill up* e *drill down*. Ad esempio la dimensione data potrebbe contenere una gerarchia del tipo Anno -> trimestre -> mese -> settimana -> data. La granularità descrive il livello di dettaglio, o di sintesi, dei dati raccolti. La scelta del giusto livello da adottare è uno step critico da affrontare durante la progettazione iniziale.

3.2.1 Modellamento concettuale

A differenza di un database relazionale, in cui l'obiettivo è di individuare delle entità e le associazioni fra esse, in un data warehouse il focus è sulla collezione di fatti, la loro misurazione e la loro contestualizzazione. Nello specifico l'obiettivo del lavoro è fornire un metodo alternativo per l'esplorazione dei dati di un'applicazione di Spark ed eventualmente il loro confronto in modo da rendere evidenti le eventuali problematiche. Si presenta una tabella dove si espongono le analisi a cui il data warehouse è in grado di rispondere.

Il primo step da affrontare è l'individuzione del fatto. In questo caso è l'evento Task ovvero il processamento di una partizione di un RDD. Questo è l'elemento avente granularità più piccola (le performance di un Executor, di un nodo, di uno Stage derivano dalle metriche dei task partecipanti a queste dimensioni). I job sono delle azioni che restituiscono un risultato e in quanto tali richiedono uno o più shuffle. Questo comporta che uno o più stage sono necessari per portare a termine il Job.

In particolare il fatto è stato modellato usando due tabelle differenti una per i Task Result e l'altra per i Task di tipo Shuffle. Le dimensioni invece sono tre: Executor, Execution Model e RDD. La dimensione Executor è utile per comprendere su quale macchina sono stati eseguiti i task; questo vuol dire che si possono aggregare le metriche e individuare la presenza di Executor che impiegano, ad esempio, più tempo nell'esecuzione rispetto ad altri. In caso di fallimento è possibile inoltre individuarne la causa grazie all'attributo *removedReason*. L'Execution Model presenta una gerarchia composta da Stage, Job e Application come rappresentato in fig. 3.2. Grazie a questa è

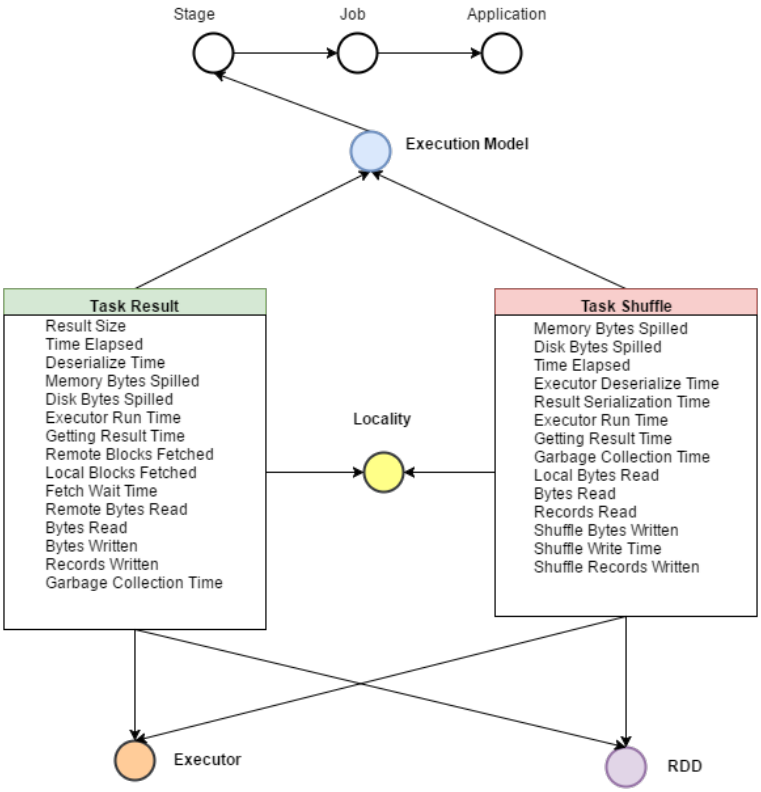

Figura 3.2: Schema concettuale

No	Analisi	Metrica
1	Numero di Task eseguiti durante l'applicazione	Count sulle fact
2	Tempo speso da un Executor, Job, Stage per fare il Garbage Collection	JvmGCTime
3	Tempo speso da un Executor, Stage, Job per la serializzazione deserializzazione	ExecDesTime
4	Tempo speso da Executor, Stage, Job per lo shuffle	ShuffleWriteTime
5	Quantità di dati trasferiti durante uno Shuffle	ShuffleBytesWritten
6	Quantità di dati che un Task scrive o legge sul Disco o RAM	MemoryBytesSpilled Disk Bytes Spilled
7	Dimensione dell'RDD	memorySize diskSize
8	Relazioni di parentela tra RDD	-
9	Quali sono gli RDD salvati in memoria e quali su disco	-
10	Numero di partizioni processate da ogni Stage Job	-

Tabella 3.1: Tipo di analisi a cui il data warehouse è in grado di rispondere

possibile ottenere l'aggregazione delle metriche e comprendere cosa accade durante ogni fase dell'esecuzione. Infine vi è la dimensione RDD: qui si tiene traccia di quali RDD e quali partizioni sono state processate; per ognuno di essi è possibile conoscere la dimensione in memoria, su disco e la relazione di parentela (padri e figli) di ognuno di essi. Per aggiungere questa caratteristica la dimensione è stata modellata come una *Slowly Changing Dimension*.

3.2.2 Schema logico

In generale il modello concettuale viene trasformato in un modello relazionale applicando un insieme di regole. In questo caso la trasformazione è stata immediata ovvero non sono stati necessari particolari accorgimenti. Nel caso specifico si è usato un modello di tipo *constellation* in cui le tabelle del fatto condividono una o più dimensioni. Si presenta adesso l'insieme delle tabelle con una descrizione dei loro attributi e del loro significato. Nella tabella 3.2 viene presentato il fatto task result. Questa tiene traccia delle

metriche relative ai task che generano un risultato. La tabella 3.3 presenta il modellamento del fatto task shuffle; questa presenta più attributi della precedente che descrivono la fase di shuffle. Per semplicità si riportano solo quelli non in comune tra le due. Nella tabella 3.4 si riporta la dimensione executor, questa riepiloga i nodi su cui i task sono stati eseguiti ed eventuali errori che comportano la rimozione dello stesso. Le tabelle 3.5-Stage, 3.6-Job, 3.7-Application formano la dimensione Execution Model ed in particolare la corrispondente gerarchia presentata nel modellamento concettuale. In particolare nell'Application sono presenti le configurazioni applicate al momento del lancio dell'applicazione; è fondamentale tenere traccia di questi parametri in quanto saranno oggetto di studio nel capitolo successivo. Infine le 3.8 e 3.9 costituiscono la dimensione RDD; queste si occupano di ricostruire la storia degli RDD creati e le loro relazioni di parentela.

TaskResult		
Nome	Significato	Pk/Fk
TaskID	Identificativo di un task	PK
AppID	Identificativo dell'applicazione	PK-FK
StageID	Referenzia lo stage cui il task appartiene	FK
ExecutorID	Referenzia l'executor in cui il task è stato eseguito	FK
Index	Partizione processata relativa ad uno specifico RDD	FK
Launch Time	Timestamp del lancio del task	-
Finish Time	Timestamp del termine del task	-
Time Elapsed	Tempo assoluto del task	-
Data Read Method	Il metodo di lettura usato. Può essere: Network, Hadoop, Disk o Memory	-
Locality	Il livello di distanza tra il task e il blocco dati	-
ExecDes Time	Tempo speso per deserializzare i dati	-
Memory Bytes Spilled	Quantità di Byte caricati in RAM	-
Disk Bytes Spilled	Bytes scritti su disco	-
Executor Run Time	Tempo speso dall'executor per il task. Include il caricamento dati dallo shuffle	-
JVMGCTime	Tempo speso dal task per il *garbage collection*	-
Remote Blocks Fetched	Numero di blocchi caricati da remoto	-
Local Blocks Fetched	Numero di blocchi locali caricati	-
Fetch Wait Time	Tempo speso per il caricamento dei blocchi dati	-
Remote Bytes Read	Byte letti da remoto	-
Local Bytes Read	Byte letti localmente	-
Total Records Read	Numero di righe lette	-
Data Write Method	Metodo di scrittura	-
Bytes Written	Byte scritti scritti in output	-
Records Written	Numero di righe scritte in output	-

Tabella 3.2: Tabella del fatto TaskResult

Task Shuffle		
Nome	Significato	Pk/Fk
Shuffle Bytes Written	Bytes scritti durante lo shuffle	-
Shuffle Write Time	Tempo speso per la scrittura durante lo shuffle	-
Shuffle Records Written	Record scritti durante lo shuffle	-

Tabella 3.3: Tabella del

Executor		
Nome	Significato	Pk/Fk
ExecID	Identificativo dell'executor	PK
AppID	Identificativo dell'applicazione	PK-FK
ExecTime	Tempo di vita dell'executor	-
ExecRemoved Reason	Codice errore	-
Host	Nome dell'Host	-

Tabella 3.4: Tabella Executor

Stage		
Nome	Significato	Pk/Fk
StageID	Identificativo dello stage	PK
AppID	Identificativo dell'applicazione	PK
JobID	Identificativo del job	FK
StageName	Nome dello stage	-
SubmissionTime	Timestamp di lancio	-
CompletionTime	Timestamp del termine	-
elapsedTime	Tempo assoluto di esecuzione	-

Tabella 3.5: Tabella Stage

Job		
Nome	Significato	Pk/Fk
AppID	Identificativo dell'applicazione	PK
PKJobID	Identificativo dell'RDD	PK
SubmissionTime	Timestamp di lancio	-
endTime	Timestamp del termine	-
elapsedTime	Tempo assoluto di esecuzione	-
Result	Flag di corretta esecuzione	-

Tabella 3.6: Tabella Job

Application		
Nome	Significato	Pk/Fk
AppID	Identificativo dell'applicazione	PK
JobID	Identificativo del job	FK
driverHost	Host che ospita il driver	-
SubmissionTime	Timestamp di lancio	-
CompletionTime	Timestamp del termine	-
elapsedTime	Tempo assoluto di esecuzione	-
appName	Nome dell'applicazione	-
driverMemory	Memoria assegnata al driver	-
execMemory	Memoria assegnata ad executor	-
rddCompress	Compressione degli RDD	-
broadcastCompress	Codec utilizzati per la compressione durante il trasferimento dati	-
storageFraction	Porzione di memoria dedicata al salvataggio dati	-

Tabella 3.7: Tabella Application

RDD		
Nome	Significato	Pk/Fk
AppID	Identificativo dell'applicazione	PK
RDDID	Identificativo dell'RDD	PK
Name	Nome dell'RDD	-
UseDisk	Flag di memorizzazione su disco	-
UseMemory	Flag di memorizzazione su RAM	-
Deserialized	Flag di serializzazione dell'RDD	-
NoOfPartition	Numero di partizioni di cui si compone l'RDD	-
NoOfCached Partition	Numero di partizioni salvate su RAM	-
MemorySize	Dimensione in memoria dell'RDD	-
DiskSize	Dimensione su disco dell'RDD	-
broadcastCompress	Codec utilizzati per la compressione durante il trasferimento dati	-

Tabella 3.8: Tabella RDD

RDD Parents		
Nome	Significato	Pk/Fk
AppID	Identificativo dell'applicazione	PK
RDDID	Identificativo dell'RDD	PK
ParentsID	Nome dell'RDD	-

Tabella 3.9: Tabella RDDParents

Capitolo 4

Best Practices e evidenze sperimentali

Per poter approfondire il funzionamento di Spark e poter tracciare delle linee guida sono stati svolti dei test. L'intero processo di test è stato condotto considerando una sola applicazione; questa è stata lanciata più volte modificando ad ogni *run* le sue impostazioni, in modo da comprendere come e quando le prestazioni migliorano. Il criterio adottato è stato quello di studiare separatamente le componenti che influiscono sull'esecuzione quali CPU e memoria assegnata, livello di locality e compressione dei dati.

Per automatizzare il processo sono stati scritti degli script in Bash[1] che leggono il file di configurazione delle applicazioni di Spark ed al termine di ognuna modificano il parametro di interesse ri-eseguendola. I risultati sono stati poi inseriti nel data warehouse presentato nel capitolo precedente grazie alla scrittura di un *parser* in Java; questo programma prende in input il file di log dell'applicazione, riorganizza le informazioni e si occupa di inserire i dati nel data warehouse. Infine i dati raccolti sono stati analizzati tramite interrogazioni SQL. Il processo è illustrato in fig. 4.1.

Tutti i test sono stati effettuati su un cluster di sei nodi aventi ognuno 2 CPU con 22 core e otto slot di RAM da 32 GB l'uno[6]. All'luopo è stato predisposto l'ambiente di calcolo Hadoop e Spark partendo dalla modifica del

[1]Bash (acronimo per bourne again shell) è una shell testuale del progetto GNU usata nei sistemi operativi Unix e Unix-like

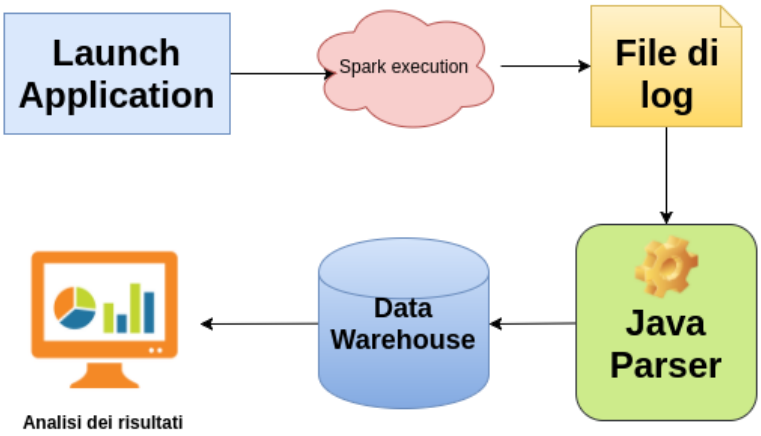

Figura 4.1: Processo di testing

file di configurazione *hosts* che ha consentito il dialogo tra le varie macchine; in seguito è stato creato il gruppo di utenza *hduser* garantendo a questo la possibilità di agire come utente di root. Si è reso inoltre necessario impostare connessioni ssh senza l'utilizzo di password per garantire allo script di installazione Cloudera la sua corretta esecuzione.

Prima di esporre i risultati è necessario fare una breve introduzione su come avviene il lancio di un'applicazione Spark e di come sia possibile migliorare (o peggiorare) le prestazioni grazie ad alcune proprietà configurabili.

4.1 Proprietà delle applicazioni Spark

La struttura di Spark fornisce la possibilità di configurare molti aspetti della vita di un'applicazione. Esistono circa 200 parametri configurabili in Spark tuttavia la maggior parte di questi non influisce sull'esecuzione delle applicazioni; ciònonostante vi è comunque un numero significativo di impostazioni che portano benefici alle performance, ad esempio la quantità di memoria e di CPU assegnate ad executor e/o il numero degli executor stessi.

Queste proprietà sono configurabili direttamente tramite lo SparkConf (argomento dello SparkContext) o tramite flag con l'uso dello spark-submit; lo spark-submit è uno script usato per il lancio di un'applicazione, ovvero è

Proprietà Spark	
Nome	Significato
spark.app.name	Nome dell'applicazione
spark.driver.memory	Memoria da assegnare al driver
spark.executor.memory	Quantità di memoria allocata per ogni executor
spark.shuffle.compress	Se comprimere l'output dei file di map

Tabella 4.1: Esempio di proprietà

un comando che prende in input il Jar contenente l'applicazione e i parametri per la sua configurazione. Il seguente codice è un esempio di utilizzo dello spark-submit in cui sono state specificate alcune semplici proprietà come il nome dell'applicazione e il parametro *spark.eventLog.enabled*.

```
./bin/spark−submit −−name "My app" −−conf spark.eventLog.enabled=false
    −−conf "spark.executor.extraJavaOptions=−XX:+PrintGCDetails
    −XX:+PrintGCTimeStamps" myApp.jar
```

Un'altro modo per specificare i parametri e quello di scriverli nel file *spark-defaults.conf*; le proprietà saranno poi caricate dinamicamente dallo script che andrà a leggerlo.

La tabella 4.1 riporta alcuni esempi di parametri configurabili.

4.1.1 Applicazione

L'applicazione che si propone è una implementazione del famoso algoritmo PageRank che assegna un peso ad ogni elemento di un collegamento web con lo scopo di quantificare la sua importanza. Questo programma è stato ottenuto dal *benchmark* HiBench[2] sviluppato da Intel il cui obiettivo è lo studio delle performance nell'ambito Big Data. HiBench si compone di un insieme di script realizzati in Bash e di applicazioni che si propongono di semplificare il processo di studio delle performance; infatti è possibile generare dataset di dimensione arbitraria e di utilizzarli come input per le applicazioni.

Questa applicazione accetta in input un file di testo formato dalle coppie <URL> e <URL vicini> e sfrutta il noto algoritmo PageRank per una clas-

Listing 4.1: PageRank app

```
1
object SparkPageRank {
3   def main(args: Array[String]) {
      if (args.length < 2) {
5       System.err.println("Usage: SparkPageRank <input_file>
            <output_filename> [<iter>]")
        System.exit(1)
7     }
      val sparkConf = new SparkConf().setAppName("ScalaPageRank")
9     val input_path = args(0)
      val output_path = args(1)
11    val iters = if (args.length > 2) args(2).toInt else 10
      val ctx = new SparkContext(sparkConf)
13
      val lines = ctx.textFile(input_path)
15    val links = lines.map{ s =>
        val elements = s.split("\\s+")
17      val parts = elements.slice(elements.length - 2, elements.length)
      }.distinct().groupByKey().persist()
19    var ranks = links.mapValues(v => 1.0)

21    for (i <- 1 to iters) {
        val contribs = links.join(ranks).values.flatMap{ case (urls, rank) =>
23        val size = urls.size
          urls.map(url => (url, rank / size))
25      }
        ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
27    }

29    val io = new IOCommon(ctx)
      io.save(output_path, ranks)
31    ranks.saveAsTextFile(output_path)

33    ctx.stop()
    }
35 }
```

sificazione dei siti web. E' importante comprendere ciò che accade durante la vita dell'applicazione: il metodo *textFile* legge il file di input e crea un numero di partizioni pari al numero di blocchi HDFS di cui si compone l'input. La funzione split crea un array separando ogni riga ad ogni spazio presente; slice seleziona un sotto insieme del file di partenza. La variabile links viene poi sottoposta ad un distinct ottenendo tutti gli URL senza duplicati, questi vengono raggruppati e infine salvati in RAM. Viene poi creata la variabile ranks associando ad ogni URL il valore 1 (in sostanza si avranno un insieme di coppie del tipo <URL, 1>). Infine viene eseguita una iterazione (configurabile come parametro del programma) in cui avviene il join tra i link creati e si calcola il peso da assegnare ad ogni sito. La fig. 4.2 mostra il DAG in cui il codice viene trasformato da Spark. Si nota in particolare la presenza di un solo Job (una sola azione) corrispondente alla linea *saveastextFile* che comporta l'esecuzione di tutte le trasformazioni. Sono presenti inoltre un totale di cinque fasi di shuffle generate rispettivamente dalle trasformazioni distinct, groupByKey e reduceByKey (quest'ultimo ripetuto un numero arbitrario di volte). Altro aspetto importante è la persistenza dell'RDD *links* grazie al comando *persist*; questo è fondamentale in quanto memorizza in RAM i dati poichè lo stesso sarà usato più volte nel corso dell'applicazione. Infine i dati di input sono stati generati utilizzando *HiBench*.

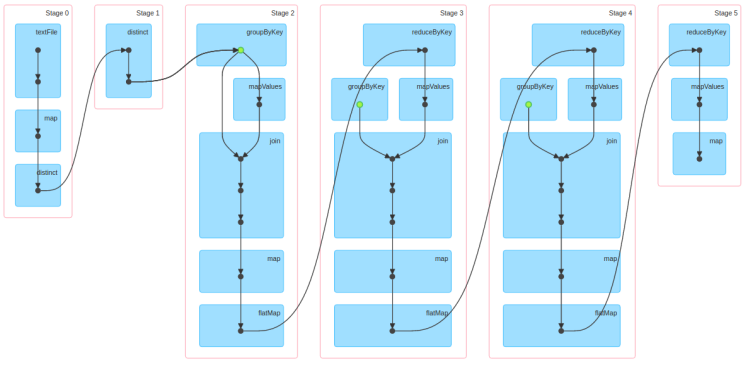

Figura 4.2: DAG dell'applicazione

4.2 Parallelismo

Uno dei fattori fondamentali per le performance di Spark è il suo livello di parallelismo; infatti un cluster non sarà pienamente usato se le operazioni non sono parallelizzate, intendo con questo la capacità di processare le partizioni in contemporanea. Il livello di parallelismo è configurabile nei seguenti modi:

- tramite le operazioni di riduzione quali groupByKey o reduceByKey che permettono di impostare il numero di partizioni. Queste comportano uno shuffle e Spark, di default, sceglie il numero di partizioni più grande tra i due RDD partecipanti allo Shuffle;

- grazie alla proprietà *spark.default.parallelism* che cambia il valore di default indicato al punto precedente;

- modificando il numero di Executor e il numero di cores assegnati ad ognuno di essi. Questo cambia la capacità di calcolo del sistema poichè consente di processare più partizioni in parallelo;

- usando la funzione coalescence che permette di ridurre il numero di partizioni senza provocare una fase di shuffle;

- usando la funzione repartition che può aumentare o diminuire il numero di partizioni provocando però uno shuffle.

Ogni executor di un'applicazione Spark ha lo stesso numero di core e la stessa quantità di ram. Il numero di cores può essere specificato tramite il flag *–executor-cores* quando si invoca lo spark-submit oppure impostando *spark.executor.cores* nel file di configurazione che controlla il numero di task in parallelo eseguibili da un executor. Analogamente la quantità di RAM può essere controllata tramite *–executor-memory* oppure con *spark.executor.memory.*

Ogni stage in Spark si compone di un numero di task, ognuno dei quali processa i dati in maniera sequenziale. Individuare il numero ideale di task è probabilmente il fattore che più incide sulle performance.

Proprietà invariate	
Numero Cores	5
Ram per executor	10 GB
Dataset di input	20 GB
Numero Partizioni	75

Tabella 4.2: Specifiche del test

Su questo tema sono stati svolti due test, uno comporta la variazione del numero di executor mentre l'altro modifica il numero di partizioni lasciando invariate altre proprietà come il numero di core assegnati ad ogni task.

4.2.1 Test su Executor

Il test è stato condotto modificando il numero di executor da un valore di partenza pari a 6 e arrivando fino a 30; si riportano in tabella 4.2 i valori delle proprietà più rilevanti che non sono state modificate tra le esecuzioni.

Successivamente sono state eseguite delle query specifiche sul data warehouse per controllare i valori di esecuzione. In fig. 4.3 vi è il tempo speso in media da ogni task per il Garbage Collection; in particolare si nota come il tempo impiegato dai task diminuisca e tenda ad assestarsi quando il numero di Executor è pari a 10.

Un altro risultato interessante è in fig. 4.4. Qui si nota nuovamente come il numero di executor ideale per l'esecuzione dell'applicazione è 10, infatti al crescere del numero di executor non diminuiscono i tempi di calcolo.

Sapendo che l'esecuzione prevede 75 partizioni e che con 10 executor (ognuno con 5 cores) si hanno un totale di 50 cores per l'intera applicazione si ottiene un valore ideale di due cores ogni tre partizioni; pertanto nel caso considerato assegnare un numero maggiore di cores a partizione non porta benefici.

In generale il primo punto da considerare per migliorare le performance è il numero di task che è possibile eseguire in parallelo. Se questo valore è troppo piccolo lo stage non è in grado di utilizzare pienamente la CPU. Per raggiungere l'ottimo bisogna infatti fare una stima di quella che è la

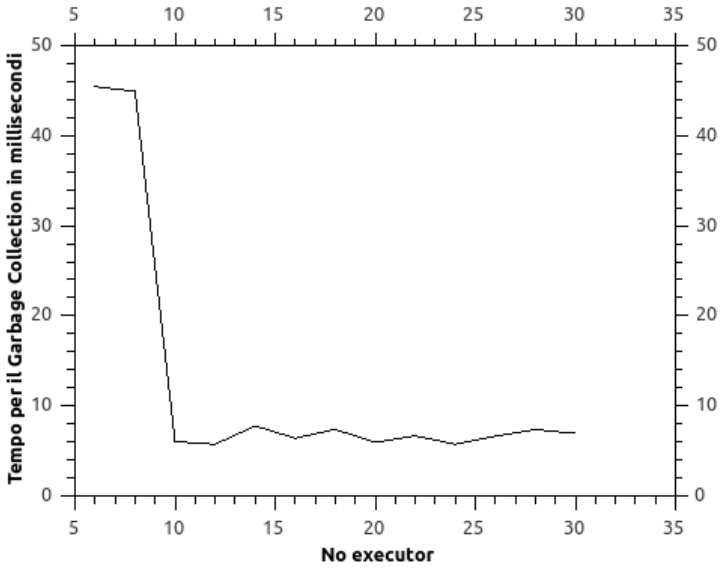

Figura 4.3: Tempo medio dei task per il GC al variare del numero di executor

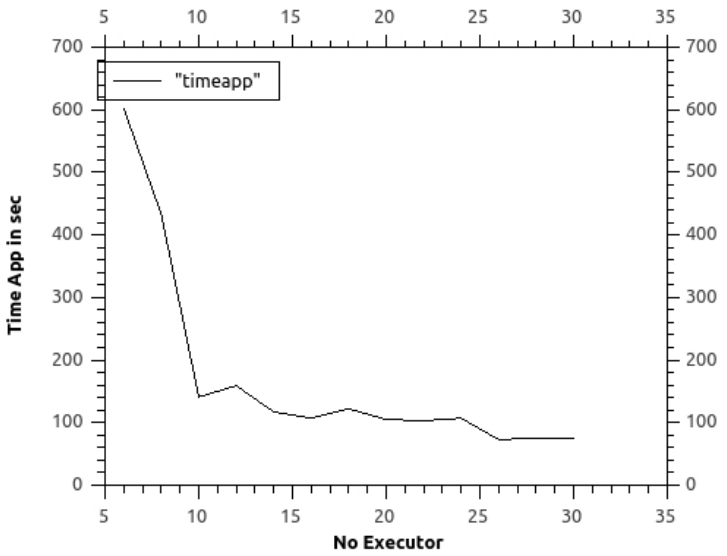

Figura 4.4: Tempo di esecuzione dell'intera applicazione al variare del numero di Executor

dimensione dell'RDD e, conoscendo il numero di cores a disposizione, cercare di assegnare 2 cores ogni 3 partizioni.

4.2.2 Test sul numero di partizioni

Si presentano adesso i risultati relativi alla variazione del parametro *spark.default.parallelism*. Questo parametro controlla il numero di partizioni che vengono generate quando si ha una fase di shuffle; nello specifico se il parametro non viene impostato, Spark sceglie come valore di default un numero di partizioni pari al più grande tra i due RDD partecipanti allo shuffle. In questo caso il numero di partizioni varia da un range di 1 fino a 17; i dati relativi all'esecuzione sono stati caricati nel data warehouse e sono stati aggregati rispetto all'intera applicazione. In fig. 4.5 si nota come il tempo di esecuzione venga dimezzato e che aumentare il numero di partizioni oltre 6 non comporta benefici.

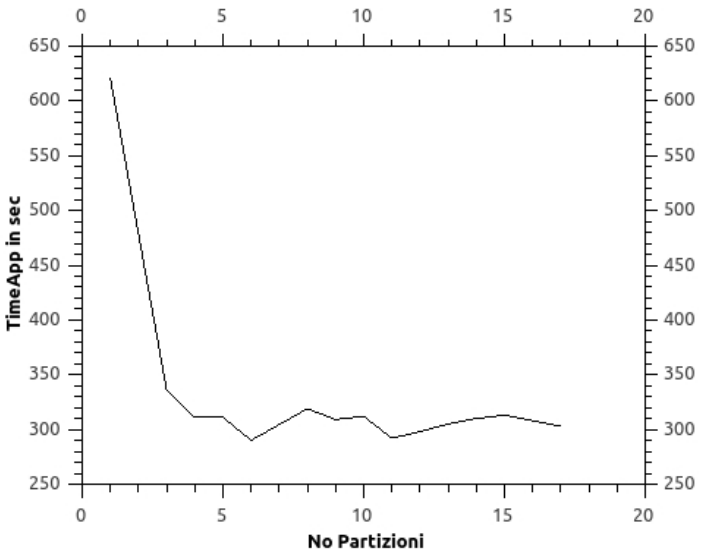

Figura 4.5: Tempo di esecuzione dell'intera applicazione al variare del numero di partizioni

In fig. 4.6 si nota una diminuzione del tempo medio di lavoro per ogni

task. Questo è chiaramente dovuto al fatto che aumentando il numero di partizioni il lavoro medio per task diminuisce.

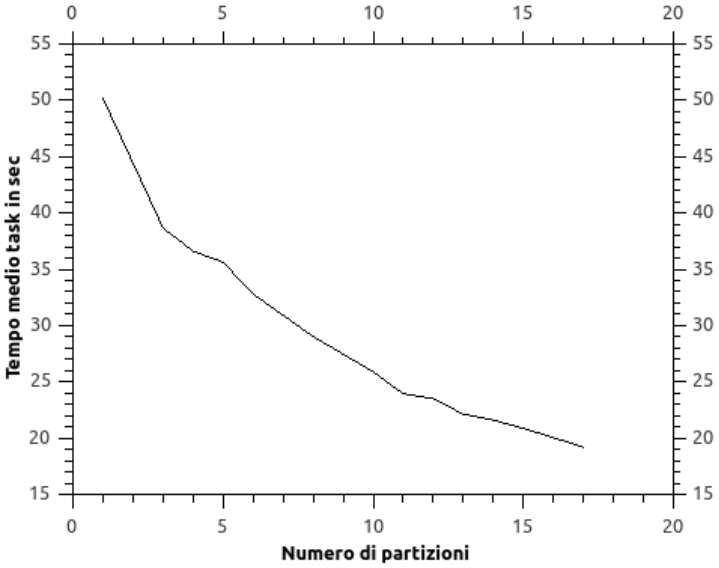

Figura 4.6: Tempo medio di esecuzione di un task

Sono stati notati anche dei miglioramenti per quanto riguarda la fase di Shuffle e per il garbage collection come mostrato rispettivamente in fig. 4.7 e fig. 4.8.

In generale ottenere il giusto grado di parallelismo nell'applicazione comporta vantaggi sotto più punti di vista. Per semplicità non vengono riportati ma il fatto di aumentare il numero di partizioni influisce sul carico di lavoro a cui un task viene sottoposto; questo conclusione deriva infatti dall'aggregazione dei dati relativi al numero di bytes letti da ogni task, infatti come aumenta il numero di partizioni così diminuisce il numero di bytes processati. Un buon modo per riuscire a determinare l'ottimo è quello di controllare il numero di partizioni nell'RDD padre e aumentarne il numero fino a quando non si migliorano le performance

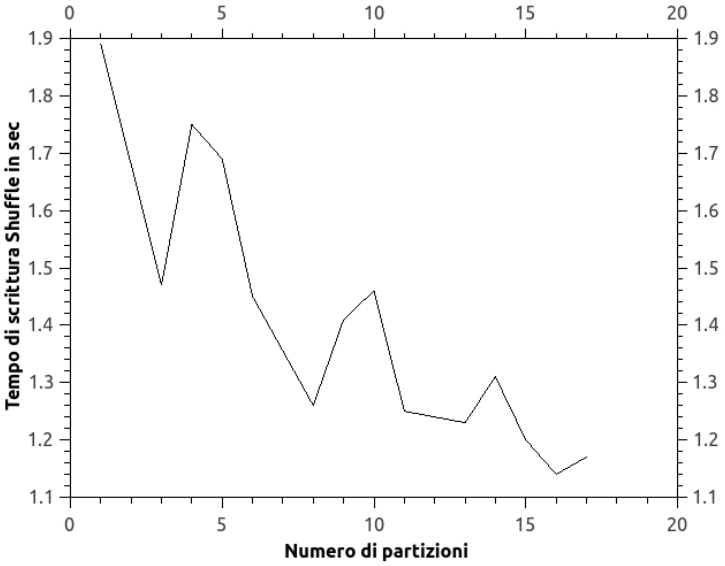

Figura 4.7: Tempo medio speso dal task per lo shuffle

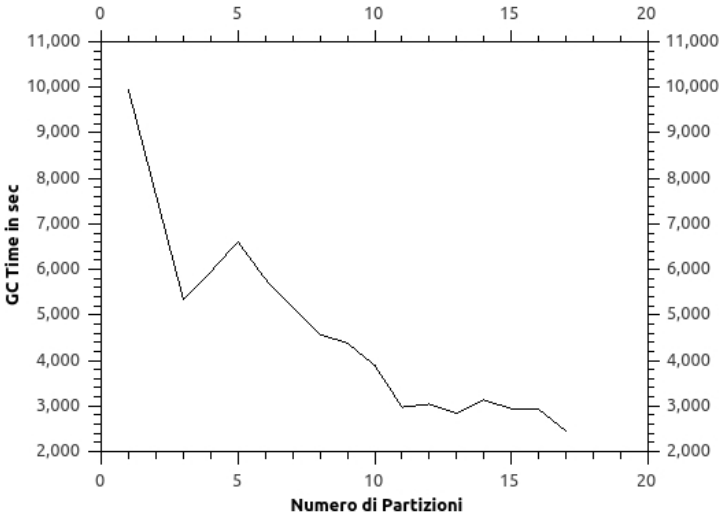

Figura 4.8: Tempo medio speso dal task per il GC

Proprietà invariate	
Numero Cores	5
Ram per executor	2 GB
Numero Executor	10
Dataset di input	20 GB
Numero Partizioni	75

Tabella 4.3: Parametri di esecuzione assegnati relativi a test memoria

4.3 Test sulla memoria

Sono state studiate le metriche di esecuzione in relazione alla modalità di gestione della memoria di Spark. Il parametro di interesse è *spark.memory.storageFraction* spiegato nel secondo capitolo. Le configurazioni dell'applicazione sono state impostate in modo da mettere in difficoltà Spark: la quantità di memoria assegnata ad executor è stata calibrata per permettere all'RDD memorizzato in cache di entrare esattamente in memoria. La motivazione è dovuta alla gestione della stessa; infatti Spark la divide in due parti: memoria d'esecuzione e memoria di storage. Quello che è possibile modificare grazie al parametro è la quantità di blocchi che non possono essere spostati dalla memoria di storage. Le proprietà invariate durante il corso dei test sono riportate in tabella 4.3.

Il parametro *spark.memory.storageFraction* è stato aumentato di 0.05 partendo da 0.1 fino ad arrivare a 0.9; si ricorda che il valore di default è pari a 0.5. Anche in questo caso le metriche sono state aggregate (calcolandone la media) rispetto alla dimensione applicazione, permettendo così un confronto tra esse. I dati che si presentano sono quindi delle medie relative al carico di lavoro sostenuto dai task indipendentemente dallo stage. La fig. 4.9 mostra il graduale miglioramento del tempo di esecuzione. E' importante sottolineare che il tempo di esecuzione quando il parametro vale 0.5 è di circa 400 secondi, quindi assegnando un valore inferiore si riscontra un peggioramento. Altri aspetti interessanti sono il tempo di scrittura di shuffle che progressivamente aumenta (fig. 4.11) e la quantità di byte che, in media, ogni task scrive su disco (fig. 4.10). Si ipotizza che l'aumento del tempo speso per la scrittura,

durante la fase di shuffle, sia dovuto alla quantità di memoria d'esecuzione che viene progressivamente diminuita: Spark tende a mettere in cache più partizioni e questo comporta risorse minori per l'operazione di shuffle. Lo stesso discorso si applica alla scrittura su disco; anche in questo caso diminuire la quantità di RAM a disposizione per lo shuffle e le operazioni di aggregazione ha portato i task a scrivere di più sul disco. Il fatto che i tempi di esecuzione siano migliorati nonostante vi sia stato un peggioramento delle altre metriche è un fenomeno che può essere spiegato dai vantaggi derivanti dalla memorizzazione dell'RDD in RAM. Si tenga presente che lo stesso RDD è processato più volte nel corso dell'applicazione e questo è un fattore che amplifica i benefici.

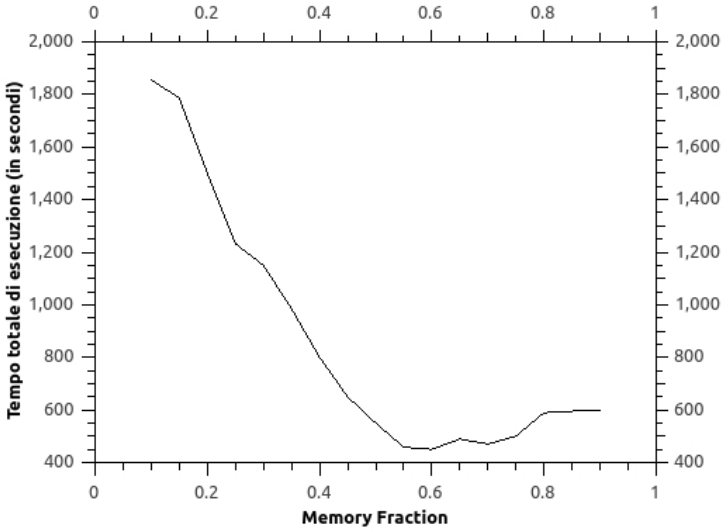

Figura 4.9: Tempo di esecuzione totale dell'applicazione

In conclusione è opportuno individuare il giusto equilibrio tra la porzione di memoria dedicata allo storage e quella per le operazioni di shuffle e aggregazione; nel caso considerato questo vale 0.6. Inoltre è importante ricordare che RDD non necessari devono essere eliminati dalla RAM sfruttando la funzione *unpersist*. Questo migliora i tempi in quanto fornisce maggiore RAM per le operazioni di aggregazione e shuffle.

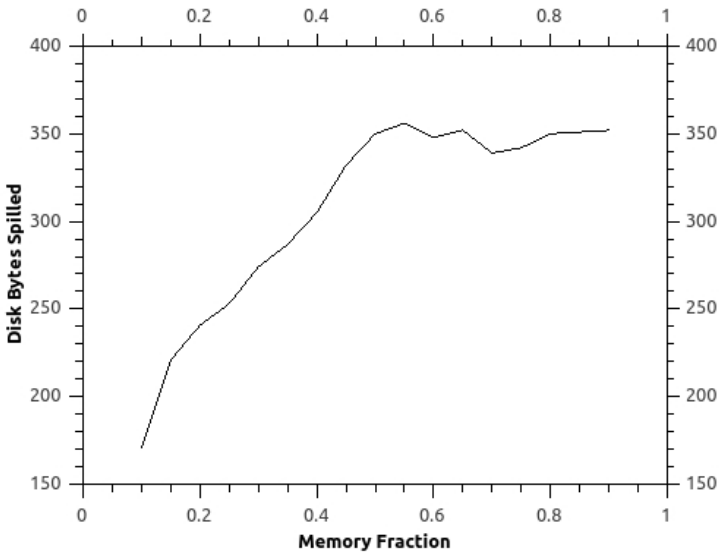

Figura 4.10: Bytes scritti su disco

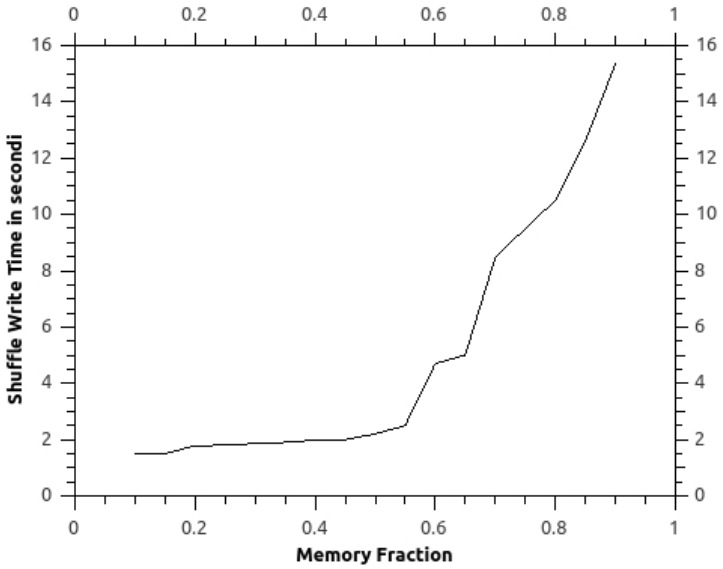

Figura 4.11: Tempo medio speso dal task per lo shuffle

Proprietà invariate	
Numero Cores	5
Ram per executor	10 GB
Dataset di input	20 GB
Numero Partizioni	75
Numero Executor	5

Tabella 4.4: Specifiche del test

4.4 Livello di storage

Quando si parla di livello di storage ci si riferisce a come Spark salva gli RDD. Esistono due funzioni in Spark che permettono di impostare come gli RDD debbano essere salvati al fine di ottimizzare i calcoli, queste sono *persist* e *cache*; entrambe hanno la stessa funzione ma *cache* permette il salvataggio solo in memoria.

L'applicazione presentata è stata modificata e ricompilata più volte modificando il livello di storage e in tabella 4.4 sono presentate le impostazioni di lancio dell'applicazione.

Sono state testate tutte e cinque le possibili configurazioni (già descritte nel capitolo 2) ma soltanto in tre di esse sono state riscontrate delle differenze significative sulle metriche. In particolare i parametri di interesse sono *Memory Deserialized*, *Memory Serialized* e *Disk Serialized*. Come nei casi precedenti i dati sono stati aggregati rispetto all'intera applicazione permettendo cosi il confronto tra le varie esecuzioni; inoltre i valori presentati sono delle medie rispetto a tutti i task eseguiti nelle singole applicazioni.

In primo luogo si osserva che salvare l'RDD in forma deserializzata comporta un tempo di esecuzione quasi doppio rispetto alle altre impostazioni (fig. 4.13). La causa principale di questo effetto sembra essere la dimensione che i dati assumono quando sono serializzati. Si è riscontrato infatti che l'RDD passa da circa 1200 MB a 148 MB; questo potrebbe a sua volta essere un fattore determinante per il tempo speso dai task nell'operazione di garbage collection (fig. 4.14) e per lo shuffle (fig. 4.12). In conclusione, nel caso preso in esame, risulta opportuno salvare i dati in forma serializzata

poichè consente di comprimere la dimensione degli RDD e questo comporta benefici sia per le operazioni di shuffle che per quelle di garbage collection.

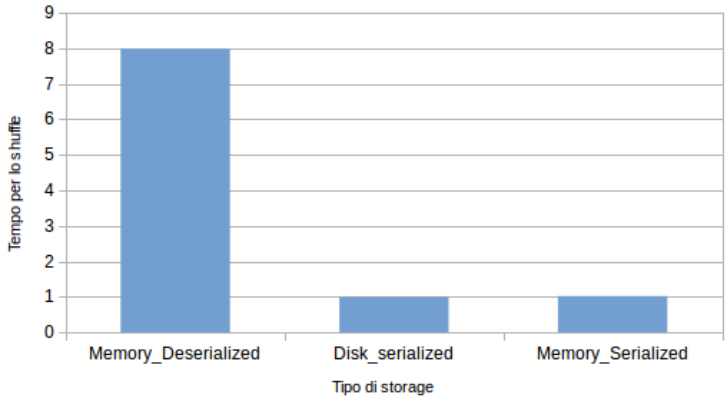

Figura 4.12: Tempo medio speso dal task per lo shuffle

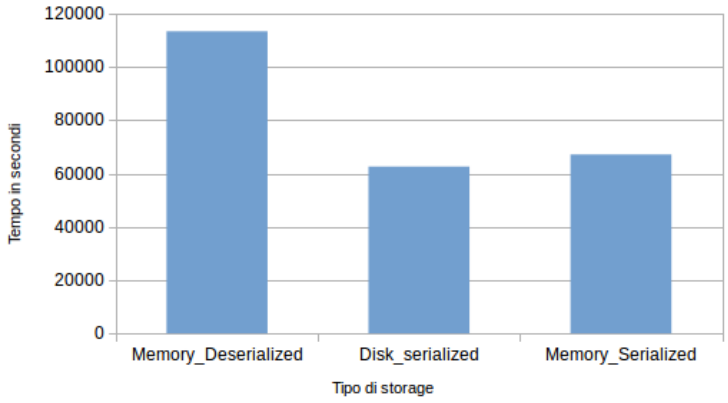

Figura 4.13: Tempo totale impiegato dell'applicazione

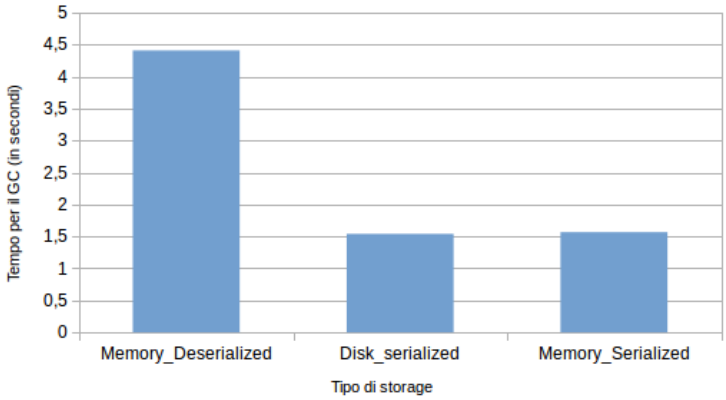

Figura 4.14: Tempo medio speso dal task per il GC

4.5 Locality

Il termine locality si riferisce alla distanza esistente tra il task e la posizione del dato da processare. Spark tenta di eseguire i task nella posizione più vicina possibile a quella dei dati in modo da minimizzare il trasferimento degli stessi. La strategia che Spark adotta prevede due opzioni: attendere fino a quando si libera la CPU per lanciare il task sullo stesso server dove i dati risiedono oppure iniziare immediatamente un nuovo task ad un livello più alto. La proprietà *spark.locality.wait* permette di configurare il tempo di attesa prima di cambiare il livello. Il cluster utilizzato si compone di sei nodi ed è quindi difficile studiare come questo fattore riesca a influire sulle performance dell'applicazione, infatti la maggior parte dei task sono eseguiti al livello più basso possibile.

I diversi livelli di località che esprimono la distanza esistente sono:

- Process Local: è il livello migliore che si possa ottenere in quanto sia il codice che i dati risiedono nella stessa JVM;
- Node Local: i dati e il codice sono sullo stesso nodo;
- Rack Local: i dati sono sullo stesso rack dei server.

Nonostante i test effettuati non abbiano evidenziato particolari risultati modificando il parametro *spark.locality.wait* è stata condotta un'analisi per

verificare come cambiano le metriche dei task a seconda del livello di località. Sono stati esplorati i dati relativi a tutte le applicazioni lanciate e si è riscontrato che il tempo medio per eseguire un task cresce come mostrato in fig. 4.15. Pertanto l'unica riflessione apprezzabile che si possa fare è che riuscire ad ottenere un livello di località di tipo process local comporta tempi di esecuzione quattro volte inferiori rispetto al livello rack local.

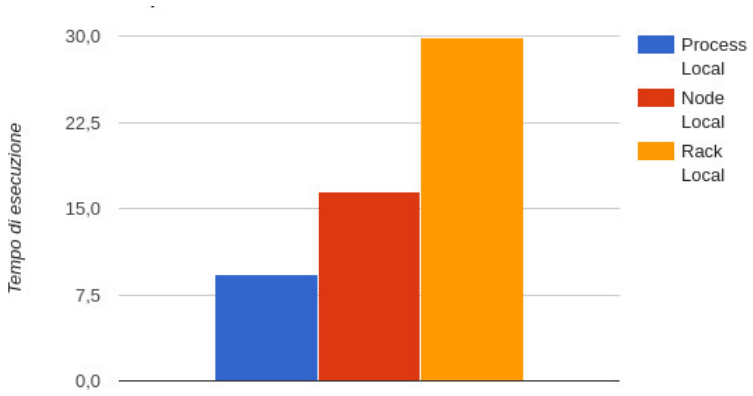

Figura 4.15: Tempo in secondi al variare del livello di locality

4.6 Utility

Al fine di poter integrare le conoscenze sulle proprietà di Spark con i vari test eseguiti è stata realizzata ad hoc una semplice utility che permette di guidare l'utente nell'allocazione delle risorse. I fattori che permettono di ottenere prestazioni migliori sono la quantità di risorse allocate ed il modo in cui Spark le gestisce. Questa utility considera questi elementi e tramite vari calcoli permette di comprendere se le risorse assegnate siano sufficienti.

I parametri in input considerati sono le proprietà spiegate nel paragrafo 4.1 e in particolare sono indicate nella tabella 4.5.

L'utente inserisce questi dati nel foglio di calcolo, il quale determina, in primo luogo, come la memoria viene ripartita tra le varie componenti.

Valori di input	
Input	Significato
dimensione del dataset	quantità di dati da salvare in memoria
spark.executor.memory	quantità di memoria da allocare ad ogni executor
num-executor	numero di executor da assegnare
spark.executor.cores	numero di core da assegnare ad ogni executor
spark.task.cpus	numero di core da assegnare per task
spark.storage.safetyFraction	percentuale di memoria allocata ad executor per evitare errori di tipo *Out Of Memory*
spark.shuffle.memoryFraction	percentuale di memoria dedicata per lo storage
block size	dimensione dei blocchi HDFS
spark.default.parallelism	numero di partizioni in cui un RDD viene suddiviso di default

Tabella 4.5: Input e relativo significato

E' noto che ogni container Yarn richiede il 10% della memoria assegnata al valore spark.executor.memory e questo è stato implementato con la seguente formula:

$$ContainerMemory = 0.1 \times executor.memory$$

La quantità di memoria effettivamente utilizzabile da ogni executor non corrisponde al parametro spark.executor.memory, questo perchè una certa quantità viene assegnata al container Yarn mentre un'altra porzione è destinata ad evitare errori di tipo *Out Of Memory*. La memoria rimanente è quindi calcolabile con la formula:

$$MemoriaUtilizzabile = (0,9 \times executor.memory) \times storage.safetyFraction$$

Il sistema a questo punto è in grado di determinare la quantità di memoria dedicata per le operazioni di caching e per quelle di shuffle conoscendo i valori impostati al lancio dell'applicazione. In formula la memoria dedicata allo storage è data da:

$$StorageMemory = MemoriaUtilizzabile \times storage.memoryFraction$$

mentre la frazione dedicata per le operazioni di shuffle e aggregazione con la formula:

$$ShuffleFraction = MemoriaUtilizzabile \times shuffle.memoryFraction$$

Noti i valori di utilizzo della memoria per ogni executor si può calcolare il quantitativo di risorse complessivamente allocate; questo viene ottenuto moltiplicando ogni singolo valore precedentemente ottenuto per il numero di executor stesso. Ad esempio viene determinato il quantitativo di RAM potenzialmente utilizzabile per il caching con:

$$StorageMemoryTot = MemoriaUtilizzabile \times storage.memoryFraction \times num-executor$$

Lo stesso discorso viene applicato per tutti gli altri parametri sopra citati. Calcolare il valore *StorageMemoryTotale* permette il confronto tra la dimensione del dataset in input con la quantità di dati che è possibile inserire in cache. Questo fornisce un'indicazione per capire se è necessario aumentare o meno le risorse allocate; in particolare un valore negativo significa che la memoria assegnata all'intera applicazione non è sufficiente. Per ovviare a questo problema si può procedere in due modi: aumentare il parametro *spark.storage.memoryFraction* o aumentare il parametro *spark.executor.memory*.

Infine oltre a questi valori lo stesso foglio di calcolo determina il parametro *"Livello di parallelismo"*. L'obiettivo principale è quello di misurare il grado di parallelismo che è possibile ottenere considerando le risorse in input fornite dall'utente; in generale è bene avere un numero di task tale che la quantità di dati da processare da ognuno di essi riesca a rientrare in memoria. Infatti un

numero basso di questi comporta che ogni task subirà una maggior pressione sulle operazioni di aggregazione (operazioni quali *groupByKey* o *join* sfuttano strutture dati *hashmap* e *buffer* per il mantenimento dei dati). La memoria disponibile ad ogni task è determinabile con:

$$TaskMemory = \frac{executor.memory \times shuffle.memoryFraction \times shuffle.safetyFraction}{executor.cores}$$

Il foglio di calcolo è in grado di determinare sia il numero di partizioni in cui l'input verrà diviso che il numero di partizioni processabili in parallelo; il primo valore è determinato con la formula:

$$NumeroPartizioni = \frac{dimensioneDataset}{PartitionSize}$$

mentre il secondo con:

$$PartizioniProcessabili = \frac{numExecutor \times spark.executor.cores}{spark.task.cpus}$$

In conclusione viene determinato il valore *"Livello di parallelismo"*; questo valore deve essere quanto più vicino al valore 0.3 in modo da assegnare 2-3 task per core, quindi se il valore risulta essere maggiore di 0.3 è necessario aumentare il numero di partizioni mentre se è inferiore si devono allocare maggiori risorse.

Dati richiesti in verde (sono presenti quelli di default)

Parametro	MB	Flag	Commento
Memoria Data ad ogni Executor	6000	spark.executor.memory	Memoria assegnata all'executor
Quantità di dati	18400		Dimensione Data Set
No Executor	7	--num-executor	Numero di Executor
No Cores per Executor	5	--executor-cores o spark.executor.cores	Numero di Cores assegnati ad ogni Executor (numero di task che un Executor può far girare in parallelo) Non assegnare più di 4/5 in quanto aumenta troppo il throghput di I/O di HDFS
No di core per task	1	spark.task.cpus	Numero di core da allocare per ogni task default 1
Parametro	**Percentuale**	**Flag**	**Commento**
spark.storage.safetyFraction	90%	spark.storage.safetyFraction	Percentuale della memoria dedicata all'executor per il suo lavoro
spark.shuffle.memoryFraction	20%	spark.shuffle.memoryFraction	Frazione dedicata allo Shuffle
spark.storage.memoryFraction	60%	spark.storage.memoryFraction	Frazione dedicata allo Storage
Calcolo le frazioni			
Frazione Shuffle	18%	spark.shuffle.memoryFraction	
Frazione cache	54%	spark.storage.memoryFraction	

Esploso Memoria di ogni Executor

Agente	Qtà RAM (MB)	Ram rimanente	Formula
Container	600	5400	0,1* Memoria Data ad ogni Executor
MemoriaUtilizzabile	4320	4320	0,9* spark.executor.memory)* spark.storage.safetyFraction
Frazione cache	2332,8	1987,2	MemoriaUtilizzabile*spark.storage.memoryFraction
Frazione Shuffle	777,6	1209,6	MemoriaUtilizzabile*spark.shuffle.memoryFraction
SafeMemory	1209,6	0	MemoriaUtilizzabile-[MemoriaUtilizzabile*(spark.storage.memoryFraction+spark.shuffle.memoryFraction)]

Figura 4.16: Foglio di calcolo

Capitolo 5

Conclusioni e sviluppi futuri

In questo lavoro di tesi è stato trattato un tema molto difficile e in continuo sviluppo, ovvero come migliorare le performance di un'applicazione Spark indipendentemente dal suo dominio applicativo. Spark permette il monitoraggio delle applicazioni e del loro funzionamento ma i parametri che variano le performance di un'applicazione sono spesso sotto valutati. E' infatti necessaria una conoscenza approfondita di questo strumento per risolvere gli eventuali problemi di inefficienza che si riscontrano come il tempo speso per il garbage collection, la quantità di dati trasferiti e un utilizzo coscenzioso ed efficiente della RAM a disposizione. Grazie alle competenze acquisite durante il corso di laurea è stato realizzato un Data Warehouse che consente all'utente di poter individuare le problematiche relative all'esecuzione di applicazioni Spark. Inoltre i test svolti hanno indagato i principali fattori che incidono sulle performance, quali il parallelismo, la gestione della memoria, il salvataggio degli RDD ed il livello di località. In particolare si sono ottenute le seguenti linee guida:

1. aumentare il numero di partizioni ottimizza il livello di parallelismo e permette di ottenere tempi di esecuzione migliori in quanto diminuisce il carico di lavoro dei singoli task;

2. risulta opportuno salvare i dati in forma serializzata se si dispone di una quantità limitata di RAM; questo perchè la compressione permette di migliorare i tempi delle operazioni di shuffle e garbage collection.

Occorre tuttavia tener presente che le operazioni di serializzazione e deserializzazione sono dispendiose in termini di utilizzo di CPU;

3. aumentare il parametro *spark.memory.storageFraction* permette di mantenere una quantità di dati maggiore in memoria a discapito delle operazioni di shuffle e aggregazione ma comunque con evidenti vantaggi nei tempi di esecuzione. Occorre valutare, però, se l'utilizzo reiterato dei dati in RAM è tale da giustificare invece una diminuizione della memoria dedicata alle operazioni sopra menzionate di shuffle e aggregazioni;

4. nel caso in cui si rilevino bassi livelli di locality è opportuno aumentare il valore *spark.locality.wait*. In questo modo si modifica il tempo che lo scheduler attende prima di passare ad un livello successivo di locality, con la riduzione ad un quarto del tempo di esecuzione medio dei task e quindi dell'intera applicazione;

5. per l'applicazione presa in esame assegnare due core ogni tre partizioni è il giusto livello di parallelismo. Dagli studi condotti non si ritiene infatti possibile determinare un modello che permetta di conoscere l'allocazione ottimale delle risorse per ogni tipologia di applicazione ma è altresì possibile sfruttare le conoscenze maturate in questo lavoro di tesi per individuare le soluzioni relative a problemi di performance.

Sviluppi futuri potranno riguardare sistemi di monitoraggio di natura grafica per l'individuazione dei colli di bottiglia e delle inefficienze; è infatti possibile sfruttare strumenti grafici con i quali il Data Warehouse può essere connesso per rendere immediatamente evidenti le inefficienze.

Bibliografia

[1] *EVERY DAY BIG DATA STATISTICS*. URL: http://www.vcloudnews.com/every-day-big-data-statistics-2-5-quintillion-bytes-of-data-created-daily/.

[2] *HiBench Suite*. URL: https://github.com/intel-hadoop/HiBench.

[3] Inmon. *Building the Data Warehouse*. 1992.

[4] *Learning Spark Lightning-Fast Big Data Analysis*. 2015.

[5] *Memory Management in Apache Spark*. URL: http://www.slideshare.net/databricks/deep-dive-memory-management-in-apache-spark.

[6] *Oracle Big Data Appliance X6-2*. URL: http://www.oracle.com/technetwork/database/bigdata-appliance/overview/bigdataappliance-datasheet-1883358.pdf.

[7] «Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computin». In: ().

[8] *Spark Internals*. URL: https://github.com/JerryLead/SparkInternals/.

[9] Tom White. *Hadoop: The Definitive Guide 4th Edition*. 2012.